Marc J. Rochkind

UNIX Programmierung für Fortgeschrittene

Systemaufrufe von access bis write
UNIX-Versionen von Berkeley bis System V
Ein- und Ausgabe von Datei bis Terminal

M. J. Rochkind
UNIX-Programmierung für Fortgeschrittene

Marc J. Rochkind

UNIX-Programmierung für Fortgeschrittene

Systemaufrufe von access bis write
UNIX-Versionen von Berkeley bis System V
Ein- und Ausgabe von Datei bis Terminal

Die deutsche Ausgabe besorgten
Rolf Olejniczak und Joachim Luiz

Eine Coedition der Verlage Carl Hanser und Prentice-Hall International

Titel der amerikanischen Originalausgabe:

Advanced UNIX programming
© 1985 by Prentice-Hall, Inc., Englewood Cliffs, New Jersey 07632

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without prior written permission of the publisher.

Alle in diesem Buch enthaltenen Programme und Verfahren wurden nach bestem Wissen und Gewissen erstellt und mit Sorgfalt getestet. Dennoch sind Fehler nicht auszuschließen. Aus diesem Grund ist das im vorliegenden Buch enthaltene Programm-Material mit keiner Verpflichtung oder Garantie irgendeiner Art verbunden. Autor und Verlag übernehmen infolgedessen keine Verantwortung und werden keine daraus folgende oder sonstige Haftung übernehmen, die auf irgendeine Art aus der Benutzung dieses Programm-Materials oder Teilen davon entsteht.

UNIX™ ist ein eingetragenes Warenzeichen der AT&T Bell Laboratories

Fachliche Beratung für den Carl Hanser Verlag:
Prof. Dr. Axel-T. Schreiner, Osnabrück

CIP-Titelaufnahme der Deutschen Bibliothek

Rochkind, Marc J.:
UNIX-Programmierung für Fortgeschrittene : Systemaufrufe von access bis write ; UNIX-Versionen von Berkeley bis System V ; Ein- u. Ausgabe von Datei bis Terminal / Marc J. Rochkind. Die dt. Ausg. besorgten Rolf Olejniczak u. Joachim Luiz. – München ; Wien : Hanser ; London : Prentice-Hall International, 1988
Einheitssacht.: Advanced UNIX programming <dt.>
ISBN 3-446-15202-4 (Hanser) Pp.
ISBN 0-13-011792-7 (Prentice Hall International) Pp.

Dieses Werk ist urheberrechtlich geschützt. Alle Rechte, auch die der Übersetzung, des Nachdruckes und der Vervielfältigung des Buches, oder Teilen daraus, vorbehalten. Kein Teil des Werkes darf ohne schriftliche Genehmigung des Verlages in irgendeiner Form (Fotokopie, Mikrofilm oder ein anderes Verfahren), auch nicht für Zwecke der Unterrichtsgestaltung, reproduziert oder unter Verwendung elektronischer Systeme verarbeitet, vervielfältigt oder verbreitet werden.

Eine Coedition der Verlage:
Carl Hanser Verlag München Wien
Prentice-Hall International Inc., London
© 1988 Prentice-Hall International Inc., London

Gesamtherstellung: Jos. C. Huber KG, Dießen/Ammersee
Printed in Germany

Inhaltsverzeichnis

Vorwort IX

Vorwort zur deutschen Ausgabe XV

1 Grundlegende Begriffe 1
- 1.1 Einführung . . . 1
- 1.2 Dateien . . . 1
  - 1.2.1 Gewöhnliche Dateien . . . 2
  - 1.2.2 Kataloge . . . 2
  - 1.2.3 Special-Files . . . 4
- 1.3 Programme und Prozesse . . . 5
- 1.4 Signale . . . 6
- 1.5 Prozeßnummern und Prozeßgruppen . . . 7
- 1.6 Zugriffsrechte . . . 9
- 1.7 Andere Prozeßattribute . . . 12
- 1.8 Prozeßkommunikation . . . 12
- 1.9 Der Gebrauch von Systemaufrufen . . . 15
- 1.10 Programmierkonventionen . . . 17
- 1.11 Portabilität . . . 19

2 Elementare Ein- und Ausgabeoperationen für Dateien 23
- 2.1 Einführung . . . 23
- 2.2 Filedeskriptoren . . . 24
- 2.3 Der Systemaufruf "creat" . . . 25
- 2.4 Der Systemaufruf "unlink" . . . 26
- 2.5 Implementierung von Semaphoren mittels Dateien . . . 27
- 2.6 Der Systemaufruf "open" . . . 30
- 2.7 Der Systemaufruf "write" . . . 35
- 2.8 Der Systemaufruf "read" . . . 39
- 2.9 Der Systemaufruf "close" . . . 39
- 2.10 Gepufferte Ein- und Ausgabe . . . 40

2.11 Der Systemaufruf "lseek" 46
2.12 Portabilität 50

3 Fortgeschrittene Ein- und Ausgabe 53
3.1 Einführung 53
3.2 Ein- und Ausgabe für Kataloge 53
3.3 Ein- und Ausgabe für Platten-Gerätedateien 56
3.4 Datum und Uhrzeit 62
3.5 Dateiattribute 66
3.6 Der Systemaufruf "link" 69
3.7 Der Systemaufruf "access" 72
3.8 Der Systemaufruf "mknod" 73
3.9 Der Systemaufruf "chmod" 75
3.10 Der Systemaufruf "chown" 76
3.11 Der Systemaufruf "utime" 77
3.12 Die Systemaufrufe "stat" und "fstat" 78
3.13 Der Systemaufruf "fcntl" 88
3.14 Portabilität 89

4 Ein- und Ausgabe für Terminals 93
4.1 Einführung 93
4.2 Normale Ein- und Ausgabe für Terminals 94
4.3 Blockierungsfreie Ein- und Ausgabe für Terminals 98
4.4 Der Systemaufruf "ioctl" 104
4.4.1 Grundlegende Anwendungen von ioctl 104
4.4.2 Geschwindigkeit, Zeichengröße und Parität 106
4.4.3 Zeichenabbildung 106
4.4.4 Verzögerungen und Tabulatorzeichen 107
4.4.5 Flußsteuerung 107
4.4.6 Steuerzeichen 107
4.4.7 Echo 108
4.4.8 Verzögerungsfreie Eingabe 109
4.5 Nackte Ein- und Ausgabe für Terminals 111
4.6 Andere Gerätedateien 113
4.7 Portabilität 113

5 Prozesse 117
5.1 Einführung 117
5.2 Das Environment 117
5.3 Die "exec"-Systemaufrufe 126
5.4 Der Systemaufruf "fork" 138
5.5 Der Systemaufruf "exit" 142

5.6 Der Systemaufruf "wait" 142
5.7 Systemaufrufe zur Bestimmung von Identifikationsnummern 146
5.8 Die Systemaufrufe "setuid" und "setgid" 147
5.9 Der Systemaufruf "setpgrp" 148
5.10 Der Systemaufruf "chdir" 148
5.11 Der Systemaufruf "chroot" 149
5.12 Der Systemaufruf "nice" 150
5.13 Portabilität 152

6 Elementare Prozeßkommunikation 155
6.1 Einführung 155
6.2 Der Systemaufruf "pipe" 156
6.3 Der Systemaufruf "dup" 162
6.4 Eine echte Shell 166
6.5 Bidirektionale Pipes 183
6.6 Portabilität 193

7 Fortgeschrittene Prozeßkommunikation 197
7.1 Einführung 197
7.2 Feinheiten eines Datenbank-Managementsystems 198
7.3 FIFO-Dateien oder Pipes mit Namen 201
7.4 Implementierung von Messages mittels FIFO-Dateien 202
7.5 Systemaufrufe zu Messages (System V) 229
7.6 Semaphore 234
7.6.1 Grundlagen zur Benutzung von Semaphoren 234
7.6.2 Implementierung von Semaphoren mittels Messages 235
7.6.3 Semaphore in System V 236
7.6.4 Semaphore in Xenix 3 240
7.7 Shared-Memory 242
7.7.1 Grundlagen zur Benutzung von Shared-Memory 242
7.7.2 Shared-Memory in System V 243
7.7.3 Shared-Memory in Xenix 3 250
7.8 Record-Locking in Xenix 3 256
7.9 Portabilität 260

8 Signale 263
8.1 Einführung 263
8.2 Signaltypen 264
8.3 Der Systemaufruf "signal" 268
8.4 Globale Sprünge 275
8.5 Der Systemaufruf "kill" 277
8.6 Der Systemaufruf "pause" 279

8.7 Der Systemaufruf "alarm" 279
8.8 Portabilität 285

9 Verschiedene Systemaufrufe 289
9.1 Einführung 289
9.2 Der Systemaufruf "ulimit" 289
9.3 Die Systemaufrufe "brk" und "sbrk" 291
9.4 Der Systemaufruf "umask" 292
9.5 Der Systemaufruf "ustat" 293
9.6 Der Systemaufruf "uname" 295
9.7 Der Systemaufruf "sync" 297
9.8 Der Systemaufruf "profil" 298
9.9 Der Systemaufruf "ptrace" 298
9.10 Der Systemaufruf "times" 299
9.11 Der Systemaufruf "time" 301
9.12 Der Systemaufruf "stime" 301
9.13 Der Systemaufruf "plock" (nur in System V) 302
9.14 Der Systemaufruf "mount" 303
9.15 Der Systemaufruf "umount" 304
9.16 Der Systemaufruf "acct" 304
9.17 Der Systemaufruf "sys3b" (nur in System V) 305
9.18 Portabilität 305

Anhang

A Prozeßattribute in System V 307

B Standard-Unterprogramme 309

Literaturverzeichnis 321

Stichwortverzeichnis 325

Vorwort

Das Thema dieses Buchs sind UNIX-Systemaufrufe, die die Schnittstelle zwischen den Benutzerprogrammen und dem UNIX-Kern bilden, auf dem sie ablaufen. Wer sich nur mit Kommandos wie etwa der Shell, Texteditoren und anderen Anwendungsprogrammen beschäftigt, muß nicht viel über Systemaufrufe wissen, für UNIX-Programmierer jedoch sind grundlegende Kenntnisse unentbehrlich. Systemaufrufe sind die *einzige* Möglichkeit, die Fähigkeiten des Kerns wie etwa das Dateisystem, die Multitasking-Mechanismen oder die elementaren Einrichtungen zur Prozeßkommunikation zu benutzen.

Eigentlich definieren die Systemaufrufe sogar die Bedeutung und den Inhalt von UNIX. Alles andere — Unterprogramm wie Kommando — ist auf diesem Grundgerüst aufgebaut. Obwohl die Neuartigkeit vieler dieser komplizierteren Programme großteils für den Ruf von UNIX verantwortlich ist, könnten diese genausogut auf jedem anderen modernen Betriebssystem programmiert worden sein. Wenn man UNIX als elegant, einfach, leistungsfähig, zuverlässig und portabel beschreibt, so bezieht man sich nicht auf die Kommandos (auf manche trifft keins dieser Attribute zu), sondern auf den Kern.

Wie schwer ist es, UNIX-Systemaufrufe zu erlernen? Als ich 1973 damit begann, in UNIX zu programmieren, war es überhaupt nicht schwierig. UNIX (und das dazugehörige Programmierer-Handbuch) bestand nur aus einem Bruchteil seiner heutigen Größe und Vielfalt. Im Handbuch gab es noch keine Programmbeispiele, dafür stand der gesamte Quellcode on-line zur Verfügung und es war einfach, Programme wie die Shell oder den Editor durchzulesen, um herauszufinden, wie Systemaufrufe funktionierten. Am wichtigsten war aber vielleicht, daß da noch erfahrene Benutzer waren, die man um Hilfe bitten konnte. Sogar Dennis Ritchie und Ken Thompson, die Erfinder von UNIX, nahmen sich Zeit für mich.

Die ehrgeizigen UNIX-Programmierer von heute sehen sich einer härteren Herausforderung gegenübergestellt als ich damals. UNIX ist mittlerweile so weit verbreitet, daß es unwahrscheinlich ist, daß ein Experte in der Nähe ist. Die meisten Computer, die unter UNIX laufen, sind nur für den Objektcode lizensiert, so daß der Quellcode von Kommandos nicht verfügbar ist. Inzwischen gibt es doppelt so viele Systemaufrufe wie 1973, und die Qualität des Handbuchs hat sich seit den

Tagen, als Ritchie und Thompson alle Systemaufrufe beschrieben, ausgesprochen verschlechtert. Das Handbuch ist nun voll von grotesken Abschnitten wie diesem:

> *If the set-user-ID mode bit of the new process file is set (see chmod (2)), exec sets the effective user ID of the new process to the owner ID of the new process file. Similarly, if the set-group-ID mode bit of the new process file is set, the effective group ID of the new process is set to the group ID of the new process file. The real user ID and the real group ID of the new process remain the same as those of the calling process.*

Als alter Hase verstand ich was damit gemeint war, als ich dies das erste Mal sah, aber ein Neuling ist sicherlich völlig verwirrt. Und bis jetzt gab es keinen Weg zurück.

Das Ziel dieses Buchs ist es, jedem erfahrenen Programmierer die Möglichkeit zu bieten, UNIX-Systemaufrufe so leicht zu lernen, wie ich es tat, und sie dann auch vernünftig und portabel einzusetzen. Deshalb ist dieses Buch mit Beispielen vollgepackt — über 3500 Zeilen C-Code. Anstatt nur Taktik (*wie* die Systemaufrufe verwendet werden) zu vermitteln, habe ich auch versucht, Strategien (*warum* und *wann* sie benützt werden) mit aufzunehmen. Außerdem gibt es eine Menge an zwanglosen Ratschlägen, die auf meiner Erfahrung an UNIX-Programmierung in den vergangenen zwölf Jahren beruhen.

Heutzutage gibt es viele UNIX-Versionen; fünf davon sind am wichtigsten:

1. System V, das neueste Angebot von AT&T. Seine Systemaufrufe bilden eine Obermenge der Systemaufrufe in System III.

2. System III, das System V voranging. (System IV wurde nicht ausgeliefert).

3. Version 7, die letzte Auslieferung, die von der Forschungsabteilung der Bell Laboratorien kam (1979). Obwohl diese Version selbst nicht mehr ausgeliefert wird, bildete sie die Grundlage für viele UNIX-Derivate, die heute vertrieben werden. Es ist vielleicht die am besten verstandene AT&T-Version, da die meisten UNIX-Bücher von Version 7 handeln.

4. Berkeley 4.2 BSD von der Universität von Kalifornien, eine umfassende Abänderung von Version 7. Viele der Systemaufrufe scheinen eine Mischung von Version 7 und System III zu sein, es gibt jedoch Dutzende von Unterschieden. Weiterhin wurden über 60 neue Systemaufrufe hinzugefügt.

5. Xenix, ein Produkt von Microsoft Corporation. Die neueste Version, die auf System III beruht, wird in diesem Buch mit Xenix 3 bezeichnet.[1] Obwohl Xenix nur eines von vielen konkurrierenden kommerziellen Systemen ist,

[1] Inzwischen gibt es Xenix 5, das auf System V beruht. A.d.Ü.

habe ich es als besonderen Schwerpunkt herausgesucht, da es die beliebteste Mikrocomputer-Version ist, und — vor allem — da sie von IBM auf dem Personal-Computer AT unterstützt wird. Dies garantiert beinahe, daß Xenix 3 auf mehr Computern laufen wird, als alle anderen Versionen zusammen.

Es ist wünschenswert, UNIX-Programme zu schreiben, die auf allen diesen fünf Versionen lauffähig sind. Soweit nicht anders angegeben, bezieht sich alles in diesem Buch auf System III, System V und Xenix 3 — und auf die zahlreichen Markennamen (wie PC/IX und UniPlus), die nur eine andere Verpackung derselben darstellen.[2] Alle Systemaufrufe aus diesen drei Versionen sind in diesem Buch enthalten, mit Ausnahme einiger unbedeutender Systemaufrufe aus Xenix 3. Ich habe außerdem Hinweise gegeben, wie man Programme schreibt, die auf Version 7 und Berkeley 4.2 BSD übertragen werden können. Die etwa 60 Systemaufrufe, die es nur in 4.2 BSD gibt, habe ich nicht mit aufgenommen, doch der größte Teil dessen, was in diesem Buch vorkommt, ist auch auf dieser Version lauffähig. Soweit Systeme wie Idris, Coherent und Regulus, die nicht auf AT&T basieren, einigermaßen gute Kopien von UNIX sind, bezieht sich dieses Buch auch auf sie, ich habe jedoch keine spezifischen Informationen mit aufgenommen. Auf jeden Fall wird Ihnen dieses Buch helfen zu lernen, wie man unter den einzelnen UNIX-Versionen programmiert — später ist es leicht, sich die Feinheiten anzueignen, wenn man einmal die Grundlagen kennt.

Wenn Sie dieses Buch lesen, sollten Sie vielleicht Ihr UNIX-Handbuch in der Nähe haben. So können Sie feststellen, ob es Unterschiede gibt zwischen dem, was ich Ihnen vorgeführt habe und der Art und Weise, wie Ihr eigenes System arbeitet. Außerdem erlaubt dies Ihnen, sich mit dem Aufbau und den Formulierungen des Handbuchs vertraut zu machen. Auf die Dauer wird das Handbuch Ihr Hauptnachschlagewerk sein, so daß Sie über kurz oder lang sowieso damit klar kommen müssen.

/usr/group, eine industrielle Vereinigung, hat einen UNIX-Standard vorgeschlagen, der auch genaue Angaben über Systemaufrufe enthält. Er stellt im wesentlichen eine Untermenge von System III (und daher von System V und Xenix 3) dar, abgesehen von dem zusätzlichen Systemaufruf **lockf** (für *Record-Locking*). Dieser Standard ist jedoch zu unvollständig, um eine Hilfe bei der Frage der Portierbarkeit zu sein — der Ausschuß konnte sich nur auf das einigen, was bei den verschiedenen Versionen sowieso bereits einheitlich war. Speziell werden Einzelheiten der Ein- und Ausgabe für Terminals und der neueren Möglichkeiten zur Prozeßkommunikation, wie *Messages*, *Semaphore* und *Shared-Memory*, ausgelassen. Überall in diesem Buch gibt es Verweise und Anmerkungen zu diesem Standard.

[2] Die starke Vermehrung der Namen rührt nicht daher, daß diese Systeme alle so verschieden sind, sondern einfach, weil es den UNIX-Wiederverkäufern verboten ist, das Warenzeichen von AT&T zu verwenden.

Dies ist ein "fortgeschrittenes" UNIX-Buch. Sie sollten bereits mit UNIX vertraut sein, zumindest als Benutzer, und Sie sollten die Programmiersprache C kennen. (Nach der Lektüre dieses Buchs werden Sie sie sogar noch besser kennen.) Kapitel 1 ist ein Auffrischungskurs, der die Fähigkeiten des UNIX-Kerns zusammenfaßt und in die Terminologie einführt, die ich fortwährend verwenden werde. Wenn Sie meinen, daß Sie Kapitel 1 unvorbereitet trifft, sollten Sie mit einem oder zwei der Bücher beginnen, die ich im Literaturverzeichnis vorgeschlagen habe.

Ich habe die Systemaufrufe in zweckmäßige Gruppen eingeteilt. Kapitel 2 bis 4 beschreiben Systemaufrufe für die Ein- und Ausgabe für Dateien und Terminals. Kapitel 5 behandelt Prozesse (Multitasking) und Kapitel 6 und 7 befassen sich mit der Kommunikation zwischen Prozessen. In Kapitel 8 werden Signale behandelt. Systemaufrufe, die sonst nirgendwo hineingepaßt haben, habe ich in Kapitel 9 aufgenommen. Anhang A stellt eine vollständige Liste der Prozeßattribute dar, und Anhang B faßt die Standard-Unterprogramme zusammen, die in den Beispielen benützt wurden. Ich habe sie für jene Leser aufgenommen, die im Bett oder im Flugzeug lesen möchten und ihr UNIX-Handbuch nicht dabei haben.

Ich habe angenommen, daß viele Leser dieses Buch nicht in der normalen Reihenfolge lesen werden, besonders jene, die bereits UNIX-Programmierer sind. Es gibt zahlreiche Querverweise, die Ihnen dabei helfen sollen, sich zurechtzufinden. Für diejenigen, die sich mit UNIX systematischer befassen wollen, bin ich in einer Reihenfolge vorgegangen, die Verweise auf noch nicht eingeführte Begriffe minimiert. Es war mir jedoch nicht möglich, derartige Verweise ganz wegzulassen, so daß auch Leser, die die Reihenfolge einhalten, manchmal die Querverweise benützen müssen. Aber in allen Fällen kann das Vorausspähen beim ersten Lesen verschoben werden, ohne daß Sie dadurch Mühe hätten, dem Text zu folgen. Ein sorgfältiges Lesen des ersten Kapitels — zu Beginn der Lektüre — wird allerdings eine beträchtliche Hilfe sein.

Dieses Buch stellt ein gutes Ergänzungslehrbuch für einen Fortgeschrittenenkurs in Betriebssystemen dar. Zusammen mit den theoretischen Prinzipien ist es für Studenten wertvoll, um zu sehen, wie die Dinge im wirklichen Leben gemacht werden. Natürlich ist das meiste von dem, was UNIX macht, schlampiger als das, was die Lehrbücher vorschlagen, aber einige Dinge über UNIX — wie seine großartige Mischung von Einfachheit und Leistungsfähigkeit — sind schwer zu verstehen, wenn man nur in Verallgemeinerungen spricht. Man muß die Einzelheiten sehen, um die Beiträge von UNIX zur Technologie von Betriebssystemen würdigen zu können.

Am Ende von Kapitel 2 bis 9 habe ich jeweils einige Aufgaben angefügt, die fast alle etwas UNIX-Programmierung erfordern. Abgesehen von einigen Ausnahmen sollten diese nur einige Stunden in Anspruch nehmen, vorausgesetzt, daß ein UNIX-System verfügbar ist und die Antwortzeiten angemessen sind. Am besten ist es, wenn diese Programme (nachdem die Fehler darin beseitigt und das Programm getestet wurde) mittels der Methode: "Lies das Programm einfach nochmal durch" durchgesehen werden. UNIX ist ergiebig genug, daß es wahrscheinlich ist, daß zwei

Programmierer zu beträchtlich verschiedenen Lösungen für das gleiche Problem gelangen. Das Teilhaben an solchen Lösungen ist mindestens so wertvoll wie das Erreichen einer einzelnen.

Obwohl ich der alleinige Autor dieses Buchs bin, habe ich vieles über UNIX nicht alleine ausgeknobelt. Die ganzen Jahre hinweg wurde ich von vielen meiner Kollegen in den Bell Laboratorien unterrichtet, besonders von Bill Burnette, Rudd Canady, Don Carter, Ted Dolotta, Alan Glasser, Rich Graveman, Dick Haight, Evan Ivie, Paul Jensen, John Linderman, Terry Lyons, John Mashey, Dennis Ritchie, Bill Roome, Ken Thompson, Larry Wehr und Peter Weinberger. Keine dieser Personen hatte irgendetwas direkt mit diesem Buch zu tun; die Ansichten — und die unvermeidlichen Fehler — stammen alle von mir.

Die genannten Personen schufen jedoch viele der Techniken und Strategien, die ich weitergegeben habe.

Brian Kernighan, der auch auf der vorigen Liste hätte stehen können, war von Anfang an in das Buch miteinbezogen. Er prüfte mein ursprüngliches Vorhaben sorgfältig, half mir weiterhin dabei, bei Prentice-Hall einen Vertrag zu sichern, und sah das endgültige Manuskript kritisch durch.

Microsoft Corporation stellte mir liebenswürdigerweise eine vollständige Sammlung an Dokumentation über Xenix 3 zur Verfügung. Dank ihrer Hilfe war ich in der Lage, Informationen des neuesten Standes aufzunehmen, die den Legionen von Programmierern, die den IBM Personal-Computer AT benützen, von großem Nutzen sein sollten.

Ich schrieb den ganzen Text und testete die meisten Programme unter PC/IX, einer Version von System III, die von Interactive Systems für IBM auf den Personal-Computer XT portiert wurde. Brian Lucas und seine Kollegen verdienen Bewunderung für das leistungsfähigste, zuverlässigste und am besten dokumentierte UNIX-System, das ich jemals benutzte. Sein Verhalten auf dem 5000 $ XT ist erstaunlich.

Ich möchte gerne Patricia Henry, meiner Verlagslektorin, und ihren Mitarbeitern bei Prentice-Hall für die schwere Arbeit danken, dieses Buch reibungslos und schnell zu veröffentlichen. Es war ein Vergnügen, mit ihnen zusammenzuarbeiten.

Schließlich gilt mein Dank Dennis Ritchie und Ken Thompson, die einer ganzen Generation von Programmierern gezeigt haben, daß Komplexität vermeidbar ist. Dutzende anderer Leute haben über die Jahre hinweg Beiträge zu UNIX geleistet, aber Dennis' und Kens zuverlässige Basis und klar dargelegte Philosophie haben bewirkt, daß UNIX immer noch das eleganteste Betriebssystem weit und breit ist.

Marc J. Rochkind
Boulder, Colorado
November 1984

Vorwort zur deutschen Ausgabe

Kennen Sie alle wichtigen und unwichtigen UNIX-Systemaufrufe? Wir waren dieser Meinung, bis wir den Systemaufruf **sys3b** in diesem Buch entdeckten (dieser gehört allerdings in die zweite Kategorie).

Marc J. Rochkind, der Entwickler des berühmten Systems zur Versionskontrolle, SCCS, zeigt in diesem Buch beispielhaft die Verwendung grundlegender Systemaufrufe zur Dateibehandlung, zum Terminalzugriff, wie auch alle Möglichkeiten der Prozeßkommunikation bis hin zur zentralisierten Datenbank. Weiterhin werden noch Signale und verschiedene nützliche Systemaufrufe durchleuchtet.

Die Systemaufrufe werden anhand von kleinen, verständlichen Beispielprogrammen vorgeführt, die Ihnen eine Verwendung in eigenen Anwendungen leichtmachen. Darüberhinaus werden Sie auch größere Programme entdecken, die die behandelten Systemaufrufe im Zusammenhang darstellen. Am Ende jedes Kapitels finden Sie einen Abschnitt, der die Portabilität der besprochenen Systemaufrufe zwischen den einzelnen UNIX-Versionen (System V, Berkeley 4.2, Version 7 und Xenix 3) behandelt.

Die vollständige Behandlung der Systemaufrufe macht dieses Buch einzigartig. Deshalb sind wir sicher, daß dieses Buch für Sie genauso wertvoll werden wird, wie es für uns bereits ist. Es sollte zum Handwerkszeug eines jeden Systemprogrammierers gehören.

Bei der Übersetzung haben wir die Formatierung der Programme unseren Schönheitsidealen angepaßt und die Programm-Kommentare ins Deutsche übersetzt. Ansonsten ließen wir die Programme (bis auf wenige Ausnahmen) unverändert. Bei Fachbegriffen haben wir uns an folgende Spielregeln gehalten:

1. Falls für den amerikanischen Begriff ein sinnvolles deutsches Äquivalent zur Verfügung stand, oder der Originalbegriff noch keinen Einzug in die deutsche UNIX-Szene gefunden hat, haben wir uns auf den deutschen Begriff geeinigt. In der Regel haben wir den Originalbegriff einmal in Klammern erwähnt. Beispiel: *Katalog* für *directory*.

2. Begriffe, bei denen keine geeignete Übersetzung möglich war oder die bereits entsprechend etabliert sind, haben wir im Original übernommen. Um den

Lesefluß nicht zu sehr zu hemmen, haben wir sie dann allerdings mit großen Anfangsbuchstaben versehen.
Beispiel: *Pipe*.

3. Bei zusammengesetzten Begriffen, die den vorigen Punkt erfüllen, haben wir Bindestriche spendiert.
Beispiel: *Message-Queue*.

4. Wir haben uns auf folgende Sprechweise (laut Duden) geeinigt: Singular: *das Semaphor*, Plural: *die Semaphore*.

Über diese Konventionen kann man geteilter Meinung sein: Falls Sie eine andere vertreten, teilen Sie sie uns mit — Sie werden dann die nächste Auflage nicht wiedererkennen. Des weiteren haben wir Kommandos, Variablen etc. **fett** gesetzt. Alle anderen wichtigen Begriffe tauchen mindestens beim ersten Mal *kursiv* auf.

Da dies ein Informatik-Buch ist, haben wir zur Produktion auch alle möglichen Hilfsmittel der modernen Informatik in Anspruch genommen. Hierbei gilt unser besonderer Dank Donald E. Knuth, dem Vater von TEX, einem hervorragenden Satzsystem, das wir zur Produktion dieser Übersetzung arg strapazierten. Weiterhin bedanken wir uns ganz herzlich bei einer Reihe von Helfern, die uns fachlich tatkräftig unterstützt haben: Guntram Blohm, Andreas Borchert, Iris Burkert, Dr. Ernst Janich, Thomas Mandry, Andreas Mehl, Klaus Stolle und Viviana Weckemann. Für die seelische Unterstützung danken wir besonders unseren Freunden und Eltern. Für die Bereitstellung der notwendigen Hard- und Software (und ca. 5000 Seiten Papier) danken wir der Firma Interplan TB Software in Ulm. Wir haben für diese Übersetzung inklusive satzreifer Formatierung ungefähr 35510400 Sekunden benötigt — der Carl Hanser Verlag, insbesondere Herr Niclas, hat hier unendliche Geduld gezeigt. Ohne Professor Dr. Axel-Tobias Schreiner, der uns zu dieser Übersetzung verleitete (was wir nicht bereuen!), der den gesamten Text sorgfältig Korrektur las und uns die ganze Zeit über mit Rat und Tat zur Seite stand, wäre diese Übersetzung nicht entstanden. Damit gilt ihm unser größter Dank.

Joachim Luiz
Rolf Olejniczak

Ulm, im Februar 1988

Kapitel 1: Grundlegende Begriffe

1.1 Einführung

Dieses Kapitel soll Ihnen einen kurzen Überblick über die Möglichkeiten geben, die der UNIX-Kern bietet. Wir wollen uns hier nicht so sehr mit den Benutzerprogrammen (Kommandos) beschäftigen, wie etwa **ls, ed** oder **sh**, die normalerweise mit dem UNIX-System ausgeliefert werden. Eine Behandlung dieser Themen fällt nicht in den Rahmen dieses Buchs. Auch das Innenleben des UNIX-Kerns (wie z.B. die Implementierung des Dateisystems) werden wir nicht genauer betrachten.

Dieser Überblick ist vielmehr als ein Auffrischungskurs gedacht. Im folgenden werden wir gewisse Ausdrücke verwenden, wie z.B. *Prozeß*, und diese erst später erklären, da wir annehmen, daß Sie ungefähr wissen, was sie bedeuten. Falls zuviel Neues für Sie dabei ist, möchten Sie vielleicht zuerst etwas vertrauter mit UNIX werden, bevor Sie weiterlesen. (Wenn Sie nicht wissen, was ein Prozeß ist, sollten Sie sich auf jeden Fall näher mit UNIX beschäftigen!) Ein gutes Buch dazu ist *The UNIX Programming Environment* von Kernighan und Pike[1]. Im Literaturverzeichnis finden Sie einige weitere Bücher, die vielleicht auch Ihren Bedürfnissen genügen. Wir gehen davon aus, daß Sie wissen, wie man in C programmiert; falls nicht, so finden Sie auch über C eine Menge Bücher.[2] [3]

1.2 Dateien

Es gibt drei Arten von UNIX-Dateien: *gewöhnliche Dateien*, *Kataloge* (*directories*) und *Special-Files* (*Gerätedateien* und *FIFO-Dateien*).

---

[1] Die deutsche Ausgabe erschien 1987 bei Hanser. A.d.Ü.

[2] Wir werden ab und zu auf das Buch *The C Programming Language* von Brian W. Kernighan und Dennis M. Ritchie verweisen, erschienen bei Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1978. Dies ist *das* Nachschlagewerk über C.

[3] Die deutsche Ausgabe erschien 1983 bei Hanser. A.d.Ü.

1.2.1 Gewöhnliche Dateien

Gewöhnliche Dateien enthalten Informationen in Form von Bytes (Zeichen), die als Vektor abgespeichert sind. Jedes einzelne Byte oder jede Folge von Bytes kann gelesen oder geschrieben werden. Lese- und Schreiboperationen beginnen bei einer Byteposition, die durch den *Positionszeiger* bestimmt wird. Dieser kann auf jede beliebige Position gesetzt werden (sogar hinter das Dateiende). Gewöhnliche Dateien werden auf Platten gespeichert.

Es ist nicht möglich, in der Mitte einer Datei Bytes einzufügen (und dabei die Datei zu dehnen) oder zu löschen (und dabei Lücken zu beseitigen). Wenn an das Ende einer Datei Bytes geschrieben werden, wächst sie: ein Byte nach dem anderen. Eine Datei kann auf die Länge Null verkürzt werden, jedoch auf keine Zwischengröße.[4] Wann immer man eine dieser nicht durchführbaren Operationen benötigt — z.B. beim Editieren von Texten — erzeugt man einfach eine ganz neue Datei. Dies ist auch sicherer.

Zwei oder mehr Prozesse können gleichzeitig die gleiche Datei lesen oder schreiben. Die Resultate hängen davon ab, in welcher Reihenfolge die einzelnen Ein- und Ausgabe-Aufrufe auftreten, und sind im allgemeinen nicht vorhersehbar. Bis vor kurzem gab es in UNIX keine leistungsfähigen Mechanismen zur Kontrolle von gleichzeitigen Zugriffen (ineffiziente Mechanismen gab es schon immer, siehe Abschnitt 2.5). Einige UNIX-Versionen stellen nun *File-Locking* und *Semaphore* zur Verfügung (siehe Kapitel 7).

Eine gewöhnliche Datei besitzt keinen Namen, sondern nur eine Nummer, die sogenannte *Inumber*. Eine Inumber ist ein Index in einen Vektor von *Inodes*, der am Anfang jeder Platte gehalten wird, die ein UNIX-Dateisystem besitzt.

Jede Inode enthält wichtige Informationen über genau eine Datei. Interessanterweise beinhaltet diese Information weder den Dateinamen noch den Dateiinhalt. Der genaue Inode-Inhalt besteht aus: Typ der Datei (gewöhnlich, Katalog oder Special-File); Anzahl Verweise (wird in Kürze erklärt); Benutzer- und Gruppennummer; drei Gruppen von Zugriffsrechten (für den Besitzer, die Gruppe und andere); Dateigröße in Bytes; Zeitpunkt des letzten Zugriffs, der letzten Änderung und der letzten Statusänderung (wann die Inode selbst zuletzt geändert wurde); und natürlich Zeiger auf Plattenblöcke, die den Inhalt der Datei enthalten.

1.2.2 Kataloge

Da es unpraktisch ist, Dateien über die Inumber anzusprechen, werden *Kataloge* unterstützt, um Namen verwenden zu können. In der Praxis wird ein Katalog fast immer dazu benützt, auf Dateien zuzugreifen. Eine Inumber wird nur zum Reparieren des Dateisystems benötigt, nachdem dieses beschädigt worden ist.

[4] In Xenix 3 ist dies durch den Systemaufruf **chsize** möglich.

Jeder Katalog besteht aus einer zweispaltigen Tabelle, mit einem Namen in der einen Spalte und der zugehörigen Inumber in der anderen Spalte. Ein Paar bestehend aus Name und Inode wird *Verweis* (*link*) genannt. Wenn der UNIX-Kern aufgefordert wird, eine Datei über einen Namen anzusprechen, so schaut er automatisch in einem Katalog nach, um die dazugehörige Inumber zu finden. Dann erhält er die zugehörige Inode, die mehr Information über die Datei enthält (z.B. wer auf sie zugreifen darf). Soll auf die Daten selbst zugegriffen werden, so steht in der Inode, wo diese auf der Platte zu finden sind.

Kataloge werden tatsächlich wie gewöhnliche Dateien gespeichert, aber sie werden in der Inode als Kataloge gekennzeichnet. Deshalb kann die Inode, die zu einem bestimmten Namen in einem Katalog gehört, die Inode eines anderen Katalogs sein. Dies erlaubt es den Benutzern, ihre Dateien in einer hierarchischen Struktur anzuordnen, für die UNIX so bekannt ist. Ein *Pfad* (*path*) wie **memo/july/smith** weist den Kern an, die Inode des *aktuellen Katalogs* (*current directory*) zu holen, um dessen Datenbytes zu bestimmen, unter diesen Datenbytes **memo** zu finden, die entsprechende Inumber zu nehmen, deren Inode zu holen, um die Datenbytes des Katalogs **memo** zu bestimmen, unter diesen **july** zu finden, die entsprechende Inumber zu nehmen, deren Inode zu holen, um die Datenbytes des Katalogs **july** zu bestimmen, **smith** zu finden und schließlich die Inode zu holen, die zu **memo/july/smith** gehört.

Woher weiß nun der Kern, wo er bei einem *relativen Pfad* (ein Pfad, der im aktuellen Katalog beginnt) anfangen muß? Er merkt sich einfach für jeden Prozeß die Inumber des aktuellen Katalogs. Ändert ein Prozeß seinen aktuellen Katalog, so muß er einen Pfad zum neuen Katalog angeben. Dieser Pfad führt zu einer Inumber, die dann als Inumber des neuen aktuellen Katalogs gespeichert wird.

Ein *absoluter Pfad* fängt mit einem / an und beginnt im *Wurzelkatalog* (*root directory*). Der Kern reserviert einfach die Inumber 2 für den Wurzelkatalog. Dies geschieht, wenn zum ersten Mal ein Dateisystem eingerichtet wird. Es gibt einen Systemaufruf,[5] um den Wurzelkatalog eines Prozesses zu ändern (in eine von 2 verschiedene Inumber), was jedoch nur sehr selten gemacht wird.

Da die Tabellenstruktur der Kataloge vom Kern direkt benutzt wird (einer der seltenen Fälle, in denen sich der Kern um den Inhalt einer Datei kümmert), und da ein fehlerhafter Katalog leicht ein ganzes UNIX-System zerstören könnte, kann kein Programm (selbst wenn es vom Super-User ausgeführt wird) in einen Katalog schreiben. Lesen dagegen darf jedes Programm, vorausgesetzt es hat die entsprechenden Privilegien dazu. Ein Programm ändert einen Katalog nur durch die Verwendung einer speziellen Menge von Systemaufrufen. Löschen und Hinzufügen eines Verweises sind schließlich die einzigen erlaubten Aktionen.

Es ist möglich, daß sich zwei oder mehr Verweise auf die gleiche Inumber beziehen, egal ob sie im selben oder in verschiedenen Katalogen stehen. Das bedeutet,

[5] **chroot**. A.d.Ü.

daß ein und dieselbe Datei mehr als einen Namen besitzen kann. Durch den Zugriff auf eine Datei über einen Pfad ergibt sich keine Mehrdeutigkeit, da nur eine Inumber gefunden wird. Diese hätte auch über einen anderen Pfad gefunden werden können, was jedoch unerheblich ist. Wird aber ein Verweis aus einem Katalog entfernt, so ist nicht sofort klar, ob die Inode und die damit verbundenen Datenbytes ebenfalls weggeworfen werden können. Deshalb enthält die Inode eine Verweiszahl. Das Entfernen eines Verweises aus einer Inode verringert nur die Verweiszahl; wenn diese Null wird, löscht der Kern die Datei wirklich.

Es gibt keinen durch die Struktur des Dateisystems bedingten Grund, warum keine mehrfachen Verweise für Kataloge erlaubt sind (wie für gewöhnliche Dateien). Es erschwert jedoch das Programmieren von Kommandos, die das gesamte Dateisystem ablaufen, so daß es der Kern verbietet.[6]

1.2.3 Special-Files

Ein *Special-File*[7] ist entweder irgendein *Gerät* (z.B. ein Magnetband oder eine Datenleitung) oder eine *FIFO*-Datei (*First-In-First-Out Queue*), die einen Mechanismus darstellt, der zur Datenübertragung zwischen Prozessen verwendet wird. Wir werden Gerätedateien hier und FIFO-Dateien in Abschnitt 1.8 behandeln.

Es gibt zwei Arten von Gerätedateien: *Block* und *Character*. *Block-Gerätedateien* richten sich nach einem bestimmten Modell: Das Gerät enthält einen Vektor von Blöcken fester Größe (üblicherweise zu je 512 Bytes), und eine Anzahl *Puffer*, die sich im Kern befinden, werden als Depot benutzt, um die Ein- und Ausgabeverarbeitung zu beschleunigen. *Character-Gerätedateien* unterliegen dagegen keinen speziellen Regeln. Sie können Ein- und Ausgabeoperationen in kleinen (einzelne Zeichen) oder in sehr großen Portionen (Plattenspuren) durchführen.

Ein und dasselbe physikalische Gerät kann sowohl Block- als auch Character-Gerätedateien besitzen, und dies trifft tatsächlich meistens bei Plattenlaufwerken zu. Gewöhnliche Dateien und Kataloge werden vom Dateisystem über eine Block-Gerätedatei angesprochen, um die Vorteile des *Blockdepots* (*buffer cache*) zu nützen. Manchmal, vorwiegend bei Datenbankanwendungen, wird direkterer Zugriff benötigt. Ein Datenbankmanager kann das gesamte Dateisystem umgehen und eine Character-Gerätedatei verwenden, um auf die Platte zuzugreifen (nicht jedoch auf den Bereich, der durch das Dateisystem belegt wird). Die meisten UNIX-Systeme besitzen zu diesem Zweck eine Character-Gerätedatei, die mittels *DMA* (*direct memory access*) Daten direkt zwischen dem Adreßraum eines Prozesses und der Platte überträgt. Dies schlägt sich in einer um ein Vielfaches höheren Leistungsfähigkeit nieder. Ein weiterer Vorteil ist die bessere Fehlerverfolgung, da man nicht den Umweg über das Blockdepot gehen muß.

[6] Der Super-User darf es. A.d.Ü.

[7] Früher, als es noch keine FIFO-Dateien gab, konnte man *Special-File* mit *Gerätedatei* gleichsetzen. A.d.Ü.

Eine Gerätedatei besitzt eine Inode, jedoch gibt es keine Datenbytes, auf die die Inode verweist. Stattdessen enthält dieser Teil der Inode eine *Gerätenummer* (*device number*). Diese stellt einen Index in eine Tabelle dar, die vom Kern benutzt wird, um eine Reihe von Unterprogrammen zu finden, die in ihrer Gesamtheit *Gerätetreiber* genannt wird.

Wenn ein Systemaufruf ausgeführt wird, um eine Operation auf einer Gerätedatei auszuführen, wird das entsprechende Gerätetreiber-Unterprogramm aufgerufen. Was dann geschieht, hängt ganz vom Innenarchitekten des Gerätetreibers ab. Da der Treiber im Kern und nicht als Benutzerprozeß abläuft, kann er auf jeden Teil des Kerns, auf jeden Benutzerprozeß und auf beliebige Register (z.B. Segmentierungsregister) oder Speicherzellen im Computer zugreifen und sie vielleicht sogar verändern. Es ist vergleichsweise leicht, neue Gerätetreiber in den Kern einzubauen, so daß man einen Henkel dafür hat, noch viel mehr zu tun als nur mit neuen Geräten zu plaudern. Dies ist der beliebteste Weg, UNIX dazu zu bringen, Dinge zu tun, die von den Entwicklern niemals vorgesehen waren. Zum Beispiel wäre es möglich, *File-* und *Record-Locking* (Sperrmechanismen auf Datei- und Satzebene) mittels eines Pseudo-Gerätetreibers zu verwirklichen, was auch schon so gemacht wurde.

1.3 Programme und Prozesse

Ein *Programm* besteht aus einer Reihe von *Anweisungen* und *Daten*, die in einer ganz gewöhnlichen Datei auf Platte gespeichert sind. Diese Datei ist in ihrer Inode als ausführbar gekennzeichnet, und der Dateiinhalt ist gemäß den vom Kern festgesetzten Regeln angeordnet. (Ein weiterer Fall, in dem sich der Kern um den Inhalt einer Datei kümmert.)

Ausführbare Dateien können von den Benutzern in beliebiger Weise erzeugt werden. Solange der Dateiinhalt den Regeln genügt und die Datei als ausführbar markiert ist, darf das Programm ausgeführt werden. Im wirklichen Leben wird meist wie folgt vorgegangen: Als erstes wird das Quellprogramm, das in irgendeiner Programmiersprache (im Normalfall natürlich C) verfaßt ist, in eine ganz gewöhnliche Datei getippt. (Solch eine Datei wird oft als *Textdatei* bezeichnet, da sie aus einzelnen Textzeilen besteht.) Als nächstes wird eine weitere gewöhnliche Datei erzeugt, *Objektdatei* genannt; diese enthält die Übersetzung des Quellprogramms in Maschinensprache. Diese Arbeit wird durch einen *Compiler* (*Übersetzer*) oder *Assemblierer* (*Assembler*) vorgenommen (die ihrerseits auch wieder Programme sind). Falls diese Objektdatei vollständig ist (d.h. es fehlen keine Unterprogramme), wird sie als ausführbar gekennzeichnet und darf in diesem Zustand ausgeführt werden.[8] Falls nicht, so wird die Objektdatei mit Hilfe eines *Binders* (*Linker*) (im UNIX-Jargon *loader* genannt) mit anderen, früher erzeugten Objektdateien zusammengebunden, die möglicherweise aus einer *Bücherei* (*library*) (Ansammlung von häufig benötigten

[8] Dies gilt seit der Einführung des Common Object File Formats in System V nicht mehr. A.d.Ü.

Objektdateien) stammen. Falls der Binder all das findet, wonach er sucht, erzeugt er eine vollständige und ausführbare Datei.

Um ein Programm auszuführen, wird der Kern zuerst angewiesen, einen neuen *Prozeß* zu generieren, d.h. eine Umgebung, in der ein *Programm* abläuft. Ein Prozeß besteht aus 3 Segmenten: *Anweisungssegment*[9] (*instruction segment*), *Benutzer-Datensegment* (*user data segment*) und *System-Datensegment* (*system data segment*). Die Programmdatei wird dazu benützt, das Anweisungssegment und das Benutzer-Datensegment zu initialisieren. Nach dieser Initialisierung gibt es zwischen einem Prozeß und dem Programm, das er ausführt, keine feste Verbindung mehr. Obwohl fortschrittliche Programmierer im allgemeinen keine sich selbst modifizierenden Programme schreiben, werden die Daten der Datensegmente natürlich verändert. Außerdem kann der Prozeß Betriebsmittel verwenden (z.B. mehr Speicher, Dateiverbindungen), die im Programm selbst nicht zur Verfügung stehen.

Es ist möglich, daß mehrere gleichzeitig ablaufende Prozesse vom gleichen Programm initialisiert werden. Zwischen diesen Prozessen besteht jedoch keine funktionale Beziehung. Der Kern könnte Speicherplatz sparen, indem diese Prozesse das Anweisungssegment gemeinsam benützen, allerdings können diese eine solche Teilung nicht erkennen, da auf die Anweisungssegmente nur Lesezugriff besteht.

Die *System-Daten* eines Prozesses beinhalten Attribute wie den aktuellen Katalog, gerade aktive Filedeskriptoren, angesammelte CPU-Zeit und so weiter. Dieses Thema wird in den nachfolgenden Abschnitten dieses Kapitels behandelt. Ein Prozeß kann auf seine System-Daten nicht direkt zugreifen oder sie verändern, da sie außerhalb seines Adreßraums liegen; stattdessen gibt es für diese Aufgabe verschiedene Systemaufrufe.

Ein neuer Prozeß wird vom Kern im Auftrag eines gerade laufenden Prozesses generiert, dieser wird der *Vater-Prozeß* des neuen *Sohn-Prozesses*. Der Sohn-Prozeß erbt die meisten System-Daten-Attribute des Vater-Prozesses. Hat z.B. der Vater irgendwelche Dateien geöffnet, so sind sie auch für den Sohn geöffnet. Erbgut dieser Art ist für die Arbeitsweise von UNIX von fundamentaler Bedeutung, wie wir im Rahmen dieses Buchs noch sehen werden.

1.4 Signale

Der Kern kann einem Prozeß ein *Signal* schicken. Ein Signal kann vom Kern selbst stammen, von einem Prozeß an sich selbst geschickt werden, von einem anderen Prozeß oder auf Veranlassung des Benutzers geschickt werden.

Der Kern signalisiert zum Beispiel eine Segmentierungsverletzung, wenn ein Prozeß versucht, auf Speicher außerhalb seines Adreßraums zuzugreifen. Ein Beispiel für ein Signal, das sich ein Prozeß selbst schickt, ist ein Wecksignal: Ein Prozeß

[9] Im UNIX-Jargon wird das Anweisungssegment *text segment* genannt, wir wollen jedoch diesen verwirrenden Begriff vermeiden.

stellt den Wecker, und wenn er abgelaufen ist, wird das Signal geschickt. Und ein Beispiel für ein Signal, das ein Prozeß einem anderen schickt, ist ein Terminierungssignal, das dann geschickt wird, wenn einer von einigen miteinander verwandten Prozessen beschließt, die ganze Familie auszulöschen. Zu guter Letzt noch ein Beispiel für ein vom Benutzer erzeugtes Signal: Ein Unterbrechungssignal wird an alle vom Benutzer erzeugten Prozesse geschickt, wenn er die *Interrupt*-Taste drückt (i.a. die DEL-Taste).

Es gibt ungefähr 19 verschiedene Typen von Signalen (einige UNIX-Versionen besitzen ein paar mehr, andere ein paar weniger). Bis auf eine Ausnahme (das *Kill*-Signal, das immer tödlich wirkt) kann ein Prozeß selbst bestimmen, was passiert, wenn er ein Signal erhält. Er kann die Standardreaktion ausführen, die eine Terminierung des Prozesses bewirkt, er kann das Signal ignorieren, oder er kann das Signal abfangen und beim Eintreffen des Signals ein Unterprogramm ausführen: Der Signaltyp (eine ganze Zahl zwischen 1 und 19) wird diesem Unterprogramm als Argument übergeben. Es ist diesem Unterprogramm jedoch nicht möglich, auf direktem Wege herauszufinden, wer das Signal geschickt hat. Wenn das Unterprogramm beendet ist, fährt der Prozeß am Unterbrechungspunkt mit der Ausführung fort.[10]

Es gibt zwei Signale, deren Bedeutung durch den Kern nicht vereinbart ist; sie können für eigene Zwecke verwendet werden.

1.5 Prozeßnummern und Prozeßgruppen

Jeder Prozeß besitzt eine *Prozeßnummer* (*process-ID*), die eine positive ganze Zahl ist. Zu jedem Zeitpunkt ist die Eindeutigkeit einer Prozeßnummer garantiert. Bis auf eine Ausnahme besitzt jeder Prozeß einen Vater. Die Ausnahme ist der Prozeß mit der Nummer 0, der durch den Kern selbst erzeugt und zum *Ein- und Auslagern von Prozessen* (*swapping*) verwendet wird.

Ein Prozeß registriert in seinen System-Daten auch die *Vater-Prozeßnummer* (*parent-process-ID*). Wird ein Prozeß durch den Tod seines Vaters zur Waise, so ändert sich seine Vater-Prozeßnummer in 1. Dies ist die Prozeßnummer des Initialisierungsprozesses (**init**), der der Urahn aller anderen Prozesse ist. Mit anderen Worten: Der Initialisierungsprozeß adoptiert alle Waisen.

Manchmal entscheiden sich Programmierer, eine größere Aufgabe als Gruppe von miteinander verwandten Prozessen anstelle eines einzelnen Prozesses zu implementieren. Zum Beispiel könnte man ein komplexes Datenbank-Managementsystem in einzelne Prozesse aufspalten, um dadurch mehr Parallelität bei der Ein- und Ausgabe auf Platte zu erlangen. Der UNIX-Kern erlaubt es, diese miteinander verwandten Prozesse als *Prozeßgruppe* (*process group*) zu organisieren.

[10] Was nicht immer wünschenswert ist. A.d.Ü.

Ein Prozeß innerhalb der Gruppe ist der sogenannte *Prozeßgruppenleiter* (*group leader*). Jedes Mitglied besitzt die Prozeßnummer des Prozeßgruppenleiters als *Prozeßgruppennummer* (*process-group-ID*).[11] [12] Der Kern stellt einen Systemaufruf zur Verfügung, mit dessen Hilfe man an jedes Mitglied einer bestimmten Prozeßgruppe ein Signal senden kann.[13] Typischerweise würde dies dazu benützt werden, eine Prozeßgruppe in ihrer Gesamtheit zu beenden, genausogut kann aber auch jedes andere Signal so gesendet werden.

Jeder Prozeß kann aus seiner Prozeßgruppe ausscheiden und der Leiter seiner eigenen (vorläufig aus einem Mitglied bestehenden) Prozeßgruppe werden, indem er seine eigene Prozeßnummer zur Prozeßgruppennummer macht und dann Sohn-Prozesse generiert, um damit seine Gruppe zu vervollständigen.[14] Somit kann z.B. ein einzelner Benutzer 10 Prozesse, zugehörig zu 3 Prozeßgruppen, ausführen.

Eine Prozeßgruppe kann ein *Kontrollterminal* (*control terminal*) besitzen. Dies ist das erste Terminal-Gerät, das vom Gruppenleiter zum Zugriff geöffnet wird.[15] [16] Normalerweise ist das Terminal, an dem sich der Benutzer anmeldet, das Kontrollterminal für alle seine Prozesse. Wird eine neue Prozeßgruppe gebildet, so besitzen die Prozesse der neuen Gruppe kein Kontrollterminal mehr.

Der Terminal-Gerätetreiber schickt *Interrupt-*, *Quit-* und *Hangup*-Signale, die von einem Terminal kommen, an alle Prozesse, die dieses Terminal als Kontrollterminal besitzen. Solange keine Vorkehrungen getroffen werden, bewirkt z.B. das Abschalten eines Terminals das Ende aller Prozesse des Benutzers. Um dies zu vermeiden, kann ein Prozeß veranlassen, daß *Hangup*-Signale an ihn ignoriert werden (genau das macht das **nohup**-Kommando).

Wenn ein Prozeßgruppenleiter aus irgendeinem Grund stirbt, so wird an alle Prozesse mit dem gleichen Kontrollterminal ein *Hangup*-Signal geschickt, das diese ebenfalls terminiert, sofern das Signal nicht abgefangen oder ignoriert wird. Diese Tatsache bewirkt, daß sich hartverdrahtete Terminals, auch wenn sie nicht abgeschaltet werden können, trotzdem so verhalten. So wird, wenn sich ein Benutzer abmeldet (also die Shell beendet, die normalerweise der Prozeßgruppenleiter ist), für den nächsten Benutzer alles aufgeräumt, gerade so, als ob der Benutzer abgeschaltet hätte.

---

[11] Nicht zu verwechseln mit der Gruppennummer eines Prozesses (process's group-ID).

[12] Mit der Gruppennummer eines Prozesses ist eine organisatorisch bedingte logische Nummer gemeint, anhand dieser die einzelnen Benutzer in sinnvolle Gruppen eingeteilt werden können. A.d.Ü.

[13] Das Kommando "kill 0" benützt diesen Systemaufruf: Alle Prozesse der Prozeßgruppe werden beendet. A.d.Ü.

[14] mit **setpgrp**. A.d.Ü.

[15] Die Handbücher von System III und System V beziehen sich manchmal auf *tty-group ID* (die Prozeßnummer des Gruppenleiters) anstatt auf *Kontrollterminal*.

[16] Dies stimmt nur, wenn noch keine Prozeßgruppe das Terminal als Kontrollterminal besitzt. A.d.Ü.

Zusammengefaßt gibt es also 3 Prozeßnummern, die zu jedem Prozeß gehören:

Prozeßnummer	Positive ganze Zahl, die einen Prozeß eindeutig identifiziert.
Vater-Prozeßnummer	Prozeßnummer des Vater-Prozesses.
Prozeßgruppennummer	Prozeßnummer des Prozeßgruppenleiters. Falls diese gleich der Prozeßnummer ist, so ist der betreffende Prozeß der Gruppenleiter.

1.6 Zugriffsrechte

Eine *Benutzernummer* (*user-ID*) ist eine positive ganze Zahl, die mit dem *Login-Namen* eines Benutzers in der *Paßwortdatei* (**/etc/passwd**) verbunden ist. Wenn sich ein Benutzer anmeldet, macht das **login**-Kommando diese Nummer zur Benutzernummer des ersten erzeugten Prozesses, der Login-Shell. Prozesse, die von der Shell abstammen, erben diese Benutzernummer.

Auch Benutzer sind in *Gruppen* (nicht zu verwechseln mit Prozeßgruppen) organisiert, die ebenfalls Nummern besitzen und mit *Gruppennummern* (*group-IDs*) bezeichnet werden. Die Login-Gruppennummer eines Benutzers wird aus der Paßwortdatei gelesen und wird zur Gruppennummer der Login-Shell.

Gruppennamen sind in einer sogenannten *Gruppendatei* (**/etc/group**) definiert. Während ein Benutzer angemeldet ist, kann er zu einer anderen Gruppe, in der er Mitglied ist, überwechseln. Dadurch wird die Gruppennummer des Prozesses geändert, der dieses Ersuchen bearbeitet (im allgemeinen die Shell, aufgefordert durch das **newgrp**-Kommando), und diese Gruppennummer wird dann an alle Nachfahren vererbt.

Die beiden obigen Nummern werden *wirkliche Benutzernummer* (*real user-ID*) und *wirkliche Gruppennummer* (*real group-ID*) genannt, da sie für den wirklichen Benutzer stehen, also die Person, die tatsächlich angemeldet ist. Es gibt noch zwei weitere Nummern für jeden Prozeß: die *effektive Benutzernummer* (*effective user-ID*) und die *effektive Gruppennummer* (*effective group-ID*). Normalerweise stimmen diese Nummern mit den entsprechenden wirklichen Nummern überein, sie können sich aber auch von ihnen unterscheiden, wie wir in Kürze sehen werden. Für den Augenblick jedoch nehmen wir an, daß die wirklichen und die effektiven Nummern gleich sind.

Die effektive Nummer wird immer dazu benutzt, Zugriffsrechte zu bestimmen; die wirkliche Nummer wird für statistische Auswertungen und zur Kommunikation unter den Benutzern verwandt. Die eine Nummer bestimmt die Zugriffsrechte des Benutzers, die andere die Identität des Benutzers.

Jede Datei (egal ob gewöhnliche Datei, Katalog oder Special-File) beinhaltet in ihrer Inode eine Benutzernummer des Besitzers (*owner user-ID*) und eine Gruppennummer des Besitzers (*owner group-ID*). Außerdem enthält die Inode drei Reihen zu je drei Bits, die den Zugriffsschutz darstellen (insgesamt also neun Bits). Jede Reihe enthält je ein Bit für *Lese-*, *Schreib-* und *Ausführungsrecht*. Ein Bit ist auf 1 gesetzt, falls das betreffende Recht erteilt ist, und auf 0, falls nicht. Je eine Reihe steht für den Besitzer, für die Gruppe des Besitzers und für alle anderen Benutzer (der Rest der Welt). Nachfolgend sind die Zuordnungen der einzelnen Bits aufgeführt (wobei Bit 0 ganz rechts steht):

Bit	*Bedeutung*	*Oktalwert*
8	Leserecht für den Besitzer	0400
7	Schreibrecht für den Besitzer	0200
6	Ausführungsrecht für den Besitzer	0100
5	Leserecht für die Gruppe	0040
4	Schreibrecht für die Gruppe	0020
3	Ausführungsrecht für die Gruppe	0010
2	Leserecht für die Welt	0004
1	Schreibrecht für die Welt	0002
0	Ausführungsrecht für die Welt	0001

Diese Bits werden häufig durch die Verwendung einer oktalen Zahl angegeben. Zum Beispiel bedeutet 775 (oktal) Lese-, Schreib- und Ausführungsrecht für den Besitzer und die Gruppe, und nur Lese- und Ausführungsrecht für die Welt. Das Kommando **ls** würde diese Kombination von Zugriffsrechten als **rwxrwxr-x** zeigen, binär dargestellt wäre es 111111101 und oktal wäre es gerade 775.

Das Zugriffsschutz-System bestimmt, ob ein Prozeß eine gewünschte Aktion (Lesen, Schreiben oder Ausführen) in Bezug auf eine bestimmte Datei durchführen kann. Bei gewöhnlichen Dateien ist die Bedeutung dieser Aktionen offensichtlich. Bei Katalogen ist die Bedeutung von "Lesen" klar, da Kataloge in ganz gewöhnlichen Dateien gespeichert sind (z.B. liest das **ls**-Kommando einen Katalog). "Schreib"-Recht bezüglich eines Katalogs bedeutet die Fähigkeit, einen Systemaufruf absetzen zu können, der einen Katalog verändern würde (einen Verweis hinzufügen oder löschen). "Ausführungs"-Recht bedeutet die Fähigkeit, den Katalog in einem Pfad verwenden zu können (manchmal auch "Such"-Recht genannt). Bezüglich Special-Files bedeuten Lese- und Schreibrecht die Fähigkeit, die Systemaufrufe **read** und **write** ausführen zu können. Was dies jedoch bewirkt, falls es überhaupt etwas bewirkt, entscheidet natürlich wieder der Entwickler des Gerätetreibers. Ausführungsrecht bei Special-Files hat keine Bedeutung.

Das Zugriffsschutz-System gewährt Zugriffsrechte nach folgendem Algorithmus:

1. Falls die effektive Benutzernummer Null ist, wird sofort Zugriff gewährt (der effektive Benutzer ist der *Super-User*).

2. Falls die effektive Benutzernummer des Prozesses mit der Benutzernummer der Datei übereinstimmt, so wird anhand der Besitzer-Bits entschieden, ob die Aktion erlaubt wird.

3. Falls die effektive Gruppennummer des Prozesses mit der Gruppennummer der Datei übereinstimmt, so wird aufgrund der Gruppen-Bits entschieden.

4. Falls weder die Benutzernummern noch die Gruppennummern übereinstimmen, dann ist der Prozeß einer vom Rest der Welt, und die dritte Reihe von Bits wird benützt.

Es gibt andere Aktionen, die man mit "Inode ändern" bezeichnen könnte, die nur der Besitzer oder der Super-User durchführen darf. Diese beinhalten das Ändern der Benutzer- oder Gruppennummer, das Ändern des Zugriffsschutzes und das Ändern der Zugriffs- und Änderungszeitstempel einer Datei. Als besonderer Fall erlaubt Schreibrecht auf eine Datei das Setzen der Zugriffs- und Änderungszeitstempel auf den augenblicklichen Zeitpunkt.

Hin und wieder möchten wir, daß ein Benutzer vorübergehend die Rechte eines anderen Benutzers bekommt. Wenn wir zum Beispiel das **passwd**-Kommando ausführen, um unser Paßwort zu ändern, hätten wir gerne, daß die effektive Benutzernummer diejenige des Benutzers **root** wird (der traditionelle Login-Name des Super-Users), da nur dieser in die Paßwortdatei schreiben darf. Dies wird dadurch erreicht, daß man **root** zum Besitzer des **passwd**-Kommandos macht (d.h. die gewöhnliche Datei, die das **passwd**-Programm enthält) und dann ein anderes Zugriffsrechte-Bit in der Inode des **passwd**-Kommandos setzt, das sogenannte *set-user-ID*-Bit. Führt man ein Programm aus, bei dem dieses Bit gesetzt ist, so wird die effektive Benutzernummer auf die des Besitzers der Datei gesetzt, in der das Programm steht. Da die effektive Benutzernummer diejenige ist, die die Zugriffsrechte bestimmt, gestattet dies einem Benutzer, vorübergehend die Zugriffsrechte eines anderen zu übernehmen. Das sogenannte *set-group-ID*-Bit wird ähnlich verwendet.

Da beide Benutzernummern (die wirkliche und die effektive) vom Vater-Prozeß an den Sohn-Prozeß vererbt werden, ist es mit dieser Methode möglich, eine effektive Benutzernummer sehr lange zu benützen.

Ein mögliches Schlupfloch gibt es aber: Sie kopieren das **sh**-Kommando in Ihren eigenen Katalog (damit werden Sie zum Besitzer der Kopie). Dann benützen Sie das **chmod**-Kommando, um das *set-user-ID*-Bit zu setzen, und das **chown**-Kommando, um den Besitzer der Datei auf **root** zu ändern. Nun führen Sie Ihre Kopie der Shell aus und übernehmen dabei die Privilegien des Benutzers **root**! Zum Glück wurde

dieses Hintertürchen schon vor langer Zeit geschlossen. Sofern Sie nicht der Super-User sind, werden beim Ändern des Besitzers einer Datei das *set-user-ID*-Bit und das *set-group-ID*-Bit automatisch gelöscht.

1.7 Andere Prozeßattribute

Neben den bereits erwähnten gibt es noch einige andere interessante Attribute, die im System-Datensegment eines Prozesses registriert sind.

Für jede von einem Prozeß geöffnete Datei (egal ob gewöhnliche Datei, Katalog oder Special-File) gibt es einen *Filedeskriptor* (eine ganze Zahl zwischen 0 und 19^{17}) und deren zwei für jede erzeugte Pipe. Ein Sohn erbt von seinem Vater-Prozeß keine Filedeskriptoren, sondern vielmehr Kopien davon. Nichtsdestoweniger sind sie beide Indizes in die gleiche systemweite Tabelle der geöffneten Dateien, was unter anderem bedeutet, daß Vater und Sohn den gleichen Positionszeiger gemeinsam haben.

Die *Priorität* eines Prozesses wird vom *Scheduler* im Kern benützt. Jeder Prozeß kann seine Priorität durch den Systemaufruf **nice** erniedrigen; ein Prozeß, der dem Super-User gehört, kann sie durch den gleichen Systemaufruf auch erhöhen. Was passiert wirklich? **nice** setzt ein Attribut, *nice-Wert* genannt, das allerdings nur einen Faktor bei der Berechnung der aktuellen Priorität ausmacht.

Die *maximale Dateigröße* (*file size limit*) eines Prozesses kann kleiner als das systemweite Limit sein (was im allgemeinen auch der Fall ist); dies soll verwirrte oder ungesittete Benutzer davon abhalten, hemmungslos anwachsende Dateien zu schreiben. Ein Prozeß des Super-Users dagegen kann sein Limit erhöhen.

Falls die *Trace*-Flagge gesetzt ist, hält ein Sohn-Prozeß bei Empfang eines Signals lediglich an, anstatt wie sonst seinen normalen Weg zu gehen (z.B. zu terminieren). Vermutlich wird dann der Vater im Adreßraum des Sohns herumstöbern und möglicherweise den Sohn neu zum Leben erwecken. Die Trace-Flagge wird nur von Debuggern (wie **adb** oder **sdb**) benützt. Dieses Hilfsmittel ist entsetzlich schwierig zu gebrauchen — glücklicherweise brauchen die meisten unter uns sich darum nicht zu kümmern, da die Debugger bereits geschrieben sind.

1.8 Prozeßkommunikation

In UNIX-Systemen vor System III konnten Prozesse nur mittels *gemeinsam benützten Positionszeigern*, *Signalen*, *Prozeß-Tracing*, *Dateien* und *Pipes* miteinander kommunizieren. Mit System III kamen *FIFO-Dateien* (benannte Pipes) und mit System V *Semaphore*, *Messages* und *Shared-Memory* hinzu. Wie wir noch sehen werden, ist keiner dieser neun Mechanismen völlig zufriedenstellend. Deshalb gibt es auch neun davon.

[17] natürlich systemabhängig. A.d.Ü.

Gemeinsam benützte Positionszeiger werden selten zur Kommunikation zwischen Prozessen verwendet. Theoretisch könnte ein Prozeß den Positionszeiger auf irgendeine gedachte Stelle in einer Datei setzen, und ein zweiter Prozeß könnte dann herausfinden, wohin dieser zeigt. Die Position (eine Zahl zwischen z.B. 0 und 100) wäre dann das übermittelte Datum. Da die Prozesse miteinander verwandt sein müssen, um sich einen Positionszeiger teilen zu können, könnten sie genauso gut auch Pipes benützen.

Signale werden manchmal benützt, wenn ein Prozeß einen anderen nur anstoßen muß. Zum Beispiel könnte ein sogenannter *Drucker-Spooler* dem aktuellen Druck-Prozeß jedesmal ein Signal geben, wenn eine neue Datei zum Drucken da ist. Für die meisten Anwendungen übergeben Signale jedoch zuwenig Daten, um hilfreich zu sein. Außerdem unterbricht ein Signal den empfangenden Prozeß und macht dadurch das Programmieren wesentlich komplizierter, als wenn der Empfänger die Mitteilung dann empfängt, wenn er auch dazu bereit ist. Hauptsächlich werden Signale dazu verwendet, Prozesse einfach nur zu beenden.

Mittels *Prozeß-Tracing* kann ein Vater-Prozeß die Ausführung seines Sohns kontrollieren. Da der Vater die Daten des Sohns lesen und schreiben kann, können die beiden ungehindert miteinander kommunizieren. Prozeß-Tracing wird nur von Debuggern benützt, da es für den allgemeinen Gebrauch viel zu kompliziert und unsicher ist. Darüber hinaus können Vater und Sohn stattdessen leicht mittels Pipes miteinander kommunizieren.

Dateien sind die am weitesten verbreitete Form der Prozeßkommunikation. Zum Beispiel könnte man in eine Datei durch einen Prozeß schreiben lassen, der **ed** ausführt, und diese dann durch einen anderen Prozeß formatieren lassen, der **nroff** ausführt. Falls zwei Prozesse jedoch gleichzeitig laufen, sind Dateien ungünstig, und zwar aus zwei Gründen: Erstens könnte der Leser schneller als der Schreiber sein, somit das Dateiende sehen und vermuten, daß die Verbindung beendet ist (dies kann man allerdings durch trickreiche Programmierung vermeiden). Zweitens wird die Datei um so größer, je länger die beiden Prozesse miteinander kommunizieren. Da Prozesse manchmal über Wochen hinweg kommunizieren, während sie Millionen von Zeichen übertragen, würde dies sehr schnell das ganze Dateisystem aufbrauchen.

Die Verwendung einer leeren Datei als Semaphor ist ebenfalls eine traditionsreiche UNIX-Technik. Sie nutzt einige Besonderheiten aus, wie unter UNIX Dateien erzeugt werden. Genaueres darüber finden Sie in Abschnitt 2.5.

Pipes lösen das Synchronisationsproblem bei Dateien. Eine Pipe ist keine spezielle Art von Datei; obwohl sie eine Inode besitzt, gibt es keine Verweise auf sie. Das Lesen und Schreiben einer Pipe ist ähnlich zum Lesen und Schreiben einer Datei, jedoch mit einigen bedeutsamen Unterschieden: Wenn der Leser den Schreiber überholt, so wartet der Leser einfach auf weitere Daten. Wenn der Schreiber dem Leser vorauseilt, pausiert er, bis der Leser eine Chance hat, ihn einzuholen, so daß der Kern nicht zuviele Daten zwischenspeichern muß. Ist schließlich ein Zeichen gelesen, so ist es für immer und ewig vorbei, so daß lang laufende Prozesse, die über

Pipes miteinander verbunden sind, das Dateisystem nicht zum Überlaufen bringen.

Pipes sind den Shell-Benutzern wohlbekannt, die Kommandozeilen wie

```
ls | wc
```

eingeben können, um zu sehen, wie viele Dateien sie besitzen. Wie wir in Kapitel 6 sehen werden, sind die Fähigkeiten des Kerns jedoch viel allgemeiner als das, was die Shell bietet.

Pipes besitzen drei größere Nachteile: Als erstes müssen die Prozesse, die über eine Pipe miteinander kommunizieren, miteinander verwandt sein, typischerweise Vater und Sohn oder zwei Geschwister. Dies zwängt viele Anwendungen zu sehr ein, besonders wenn ein Prozeß ein Datenbankmanager und der andere ein Anwender ist, der Zugriff auf die Datenbank benötigt. Der zweite Nachteil besteht darin, daß nicht garantiert ist, ob Lese- und Schreiboperationen atomar sind, und somit der Gebrauch von Pipes im Falle von mehreren Lese- oder Schreib-Prozessen verhindert wird.[18] Wenn mehrere Prozesse Anfragen an den Datenbankmanager richten, könnte es passieren, daß ihre Daten vermischt werden. Der dritte Nachteil liegt darin, daß Pipes möglicherweise zu langsam sind: Daten müssen vom schreibenden Benutzer-Prozeß zum Kern und von dort zurück zum Lese-Prozeß kopiert werden. Obwohl keine wirkliche Ein- und Ausgabe durchgeführt wird, kann allein das Kopieren für manche kritische Anwendungen zu lange dauern. Aufgrund dieser Nachteile wurden phantasievollere Pläne entworfen.

In System III wurden *FIFO-Dateien* oder *benannte Pipes* hinzugefügt, um die beiden ersten Nachteile der Pipes zu beheben. (FIFO steht für "First-In-First-Out".) Eine FIFO-Datei existiert wie ein Special-File, und jeder dazu berechtigte Prozeß darf sie zum Lesen oder Schreiben öffnen. Unteilbarkeit ist garantiert: Die Zeichen, die durch einen einzelnen Systemaufruf gelesen oder geschrieben werden, sind immer zusammenhängend. Folglich läßt sich die Verständigung sowohl zwischen mehrfachen Lese- als auch mehrfachen Schreib-Prozessen leicht bewältigen. Wie wir in Kapitel 7 zeigen werden, läßt es sich mit FIFO-Dateien auch leicht programmieren.

Was ist nun bei den FIFO-Dateien nicht in Ordnung? Nun, auch sie beseitigen den dritten Nachteil von Pipes nicht: Sie sind manchmal zu langsam. Für die kritischsten Anwendungen können die neuen Prozeßkommunikationsmöglichkeiten von System V benützt werden, jedoch ist dies nicht ganz einfach.

Ein *Semaphor* ist eine Flagge, die zwei oder mehr Prozesse daran hindert, auf die gleichen Betriebsmittel zur gleichen Zeit zuzugreifen (der Begriff kommt aus dem Eisenbahnwesen, wo die "Betriebsmittel" ein gemeinsames Gleis und die "Prozesse" Züge waren). Wie wir bereits erwähnten, können auch Dateien als Semaphore verwendet werden, aber für viele Applikationen ist dies unnötiger Aufwand.

[18] In neueren UNIX-Versionen besteht dieses Problem nicht mehr.

Eine *Message* ist eine kleine Datenmenge (von z.B. 500 Bytes), die an eine *Message-Queue* (*Nachrichten-Schlange*) geschickt werden kann. Jeder Prozeß mit den passenden Rechten kann Messages aus einer Message-Queue erhalten. Er hat dabei die Wahl unter mehreren Möglichkeiten: die erste Message, die erste Message eines bestimmten Typs oder die erste Message einer Gruppe von Typen.

Shared-Memory (*gemeinsame Speicherbereiche*) bietet die schnellste Art der Prozeßkommunikation. Ein und derselbe Speicherbereich wird über den Adreßraum von zwei oder mehr Prozessen gelegt. Sobald Daten in das Shared-Memory geschrieben werden, stehen sie den Lese-Prozessen sofort zur Verfügung. Mittels eines Semaphors oder einer Message werden Lese- und Schreib-Prozeß synchronisiert.

Auch unter Xenix 3 gibt es Semaphore und Shared-Memory.

1.9 Der Gebrauch von Systemaufrufen

Wie setzt eigentlich ein C-Programmierer einen Systemaufruf ab? Nun, es besteht kein Unterschied zwischen einem Systemaufruf und einem beliebigen anderen Funktionsaufruf. Zum Beispiel könnte man den **read**-Systemaufruf so benutzen:

```
amt = read(fd, buf, numbyte);
```

Die Implementierung des Unterprogramms **read** ist je nach Art der UNIX-Implementierung verschieden. Im allgemeinen besteht sie aus einem Assemblerprogramm, das einen Maschinenbefehl benützt, der speziell für Systemaufrufe entwickelt wurde und nicht direkt von C aus ausführbar ist. Dieser Befehl kann, abhängig vom jeweiligen Computer, "sys", "int", "svc" etc. heißen.

Im UNIX-Handbuch wurden früher einmal Systemaufrufe sowohl in Assembler als auch in C beschrieben; die Assembler-Dokumentation wurde jedoch weggelassen, als UNIX auf andere Maschinen als die PDP-11 portiert wurde. Heutzutage kann man mit ziemlicher Sicherheit annehmen, daß Systemaufrufe einfach nur C-Unterprogramme sind. Bedenken Sie jedoch, daß ein Systemaufruf viel länger dauert als nur ein einfacher Unterprogrammaufruf innerhalb des prozeßeigenen Adreßraums, da ein Systemaufruf einen sogenannten *Context-Switch* (vom Benutzer zum Kern und zurück) bewirkt.[19] Deshalb ist es klug, überflüssige Systemaufrufe zu vermeiden. Dieser Punkt wird in Abschnitt 2.10 vertieft, wenn wir gepufferte Ein- und Ausgabe untersuchen.

Die meisten Systemaufrufe liefern einen Wert zurück. Im obigen Beispiel von **read** wird die Anzahl gelesener Zeichen zurückgeliefert. Um einen Fehler anzuzeigen liefert ein Systemaufruf einen Wert, der nicht als zulässiges Resultat mißverstanden werden kann, nämlich meistens -1. Deshalb hätte unser **read**-Beispiel etwa wie folgt codiert sein müssen:

[19] Der Kern führt den Systemaufruf im Auftrag des Prozesses aus. A.d.Ü.

```
if ((amt = read(fd, buf, numbyte)) == -1)
{
    printf("Read failed!\n");
    exit(1);
}
```

Beachten Sie, daß auch **exit** ein Systemaufruf ist, der jedoch keinen Fehler zurückliefern kann.

Es gibt viele Gründe, weshalb ein Systemaufruf, der -1 zurückliefert, schiefgegangen sein könnte. Die globale Integer-Variable **errno** enthält einen Code, der diesen Grund anzeigt. Im Unix-Handbuch sind diese Fehler-Codes am Anfang jenes Kapitels erklärt, das Systemaufrufe behandelt (die Seiten, die mit "intro(2)" betitelt sind). Beachten Sie, daß **errno** nur dann eine gültige Angabe enthält, wenn ein Systemaufruf tatsächlich -1 zurückliefert; Sie können allein mit **errno** nicht herausfinden, ob ein Fehler aufgetreten ist.

Es gibt noch zwei andere nützliche externe Variablen: **sys_errlist** ist ein Vektor von Fehlermeldungen, die den Fehler-Codes entsprechen, und **sys_nerr** ist eine um eins höhere Zahl als der höchste definierte Vektorindex für **sys_errlist**.[20] Bevor man also mit **errno** auf **sys_errlist** zugreift, sollte man vorher **errno** anhand von **sys_nerr** auf Zulässigkeit testen. Nun sind wir in der Lage, ein nützliches Unterprogramm zu schreiben, das wir das ganze Buch hindurch immer dann verwenden wollen, wenn ein Systemaufruf einen Fehler liefert:

```
#include <stdio.h>

void syserr(msg)  /* Fehlermeldung ausgeben und terminieren */
    char * msg;
{
    extern int errno, sys_nerr;
    extern char * sys_errlist[];

    fprintf(stderr, "ERROR: %s (%d", msg, errno);
    if (errno > 0 && errno < sys_nerr)
        fprintf(stderr, "; %s)\n", sys_errlist[errno]);
    else
        fprintf(stderr, ")\n");
    exit(1);
}
```

[20] Die meisten UNIX-Handbücher definieren sys_nerr als die *Anzahl* (Fehler-)Meldungen, was jedoch falsch ist, da es keine Meldung mit der Nummer Null gibt.

Wir könnten **syserr** dann wie folgt verwenden:

```
if ((amt = read(fd, buf, numbyte)) == -1)
    syserr("read");
```

Da wir vergessen haben, den Filedeskriptor **fd** zu öffnen, erhielten wir folgende Nachricht, als wir das Programmstück ausführten:

```
ERROR: read (9; Bad file number)
```

Diese Nachricht ist nicht besonders ansprechend, aber für unsere Zwecke ist sie gerade recht. Sie können sicherlich unsere Version von **syserr** verbessern, wenn Sie anfangen, Ihre eigenen Programme zu schreiben. Ja, vielleicht möchten Sie gar nicht, daß eine Routine, die die Fehler bei einem Systemaufruf behandelt, zum Programmende führt. Für ein Unterprogramm ist es oft besser, einen Fehler-Code an seinen Aufrufer zurückzugeben (der ihn dann seinerseits an *seinen* Aufrufer weiterreichen könnte, etc.), so daß korrigierende Maßnahmen ergriffen werden können.

Hier eine weitere Funktion, die wir von Zeit zu Zeit gebrauchen werden, um verhängnisvolle Fehler zu behandeln, die nicht von fehlgegangenen Systemaufrufen herrühren:

```
#include <stdio.h>

void fatal(msg)  /* Fehlermeldung ausgeben und terminieren */
    char * msg;
{
    fprintf(stderr, "ERROR: %s\n", msg);
    exit(1);
}
```

1.10 Programmierkonventionen

In den C-Beispielen dieses Buchs kommen einige Sprachbestandteile vor, die erst nach dem Erscheinen von *The C Programming Language* hinzugefügt wurden, nämlich die Datentypen **enum** und **void**.

Ein Objekt vom Typ **enum** ist wie eine Integer, mit dem Unterschied, daß es nur einen Wert annehmen kann, der in seiner Vereinbarung aufgezählt worden ist (diese Idee stammt von Pascal). Dazu ein Beispiel:

```
enum color {RED, GREEN, BLUE} balloon;
```

Hier wird ein Aufzähltyp **color** vereinbart und weiterhin eine Variable **balloon** von diesem Typ. **RED**, **GREEN** und **BLUE** werden auch als Konstanten vom Typ **color** vereinbart; **balloon** kann einen dieser drei Werte annehmen.

void stellt einen Typ dar, der nur bei Funktionen anwendbar ist. Er gibt an, daß eine Funktion *keinen* Wert liefert — vergleichbar einer Prozedur in Pascal oder einem Unterprogramm in Fortran.

An dieser Stelle ist es angebracht, einige weitere Bemerkungen über unsere Programmier-Konventionen zu machen. In keinem unserer Beispiele taucht eine **register**-Vereinbarung auf. (Eine **register**-Vereinbarung ist eine Bitte an den Compiler, eine Variable der Speicherklasse **auto** an ein Hardwareregister zuzuweisen.) Wir haben sie weggelassen, da sie die Bedeutung einer Funktion nicht beeinflußt, und wir die Beispiele so einfach wie möglich halten wollen. Der aufmerksame Leser wird sicher viele Wege finden, wie unsere Beispiele schneller oder kleiner gemacht werden könnten. Häufig opfern wir Effizienz zu Gunsten von Klarheit und Zuverlässigkeit. Da dies andererseits auch ein Buch und kein Computer ist, ist Effizienz wahrscheinlich als "Klarheit und Zuverlässigkeit" definiert.

In ein paar Beispielen, insbesondere in Kapitel 4, werden wir Methoden zum Abfragen und Setzen von Bits benützen, mit denen einige Leser womöglich nicht so vertraut sind, obwohl die verwendeten C-Operatoren nicht neu sind (vielleicht sollten Sie sie in Ihrem C-Handbuch noch einmal ansehen). Im folgenden Beispiel definiert **MASK** zwei Bits. Wir fragen die Variable **flags** ab, um nachzusehen, ob diese gesetzt sind. Falls sie gesetzt sind, löschen wir sie, falls nicht, so setzen wir sie.

```
#define MASK 0201

fixflags()
{
    int flags;

    if ((flags & MASK) == MASK)  /* falls gesetzt ... */
        flags &= ~MASK;          /* ... loeschen      */
    else
        flags |= MASK;           /* sonst setzen      */
}
```

Es ist wichtig, zu verstehen, daß

```
if (flags & MASK)                /* falsch            */
```

falsch gewesen wäre. Wenn nur eines der beiden Bits gesetzt wäre, würde der Ausdruck **flags & MASK** immer noch ungleich Null (also wahr) ergeben. Wenn

die Maske nur aus einem Bit bestanden hätte, wäre diese einfachere **if**-Anweisung korrekt gewesen, aber es wäre eine schlechte Technik gewesen, da man für **MASK** keinen bestimmten Wert voraussetzen sollte.

Viele Beispielfunktionen in diesem Buch liefern einen Wert vom Typ **BOOLEAN**, entweder **TRUE** oder **FALSE**. Dieser Typ ist unserer Definitionsdatei **defs.h** vereinbart. Dort sind auch zwei praktische Makros definiert, mit denen man das höchste (am weitesten links stehende) und das niedrigste (am weitesten rechts stehende) Byte eines 16-Bit Worts erhält:[21]

```
typedef enum {FALSE, TRUE} BOOLEAN;

#define lowbyte(w)  ((w) & 0377)
#define highbyte(w) lowbyte((w) >> 8)
```

Einige Systemaufrufe benötigen eine Definitionsdatei, die Konstanten und Strukturen vereinbart. Wir werden diese erwähnen, wenn wir solche Systemaufrufe benutzen. Da ein paar Definitionsdateien sehr häufig verwendet werden, wollen wir die entsprechenden **#include**-Anweisungen nicht in jedem Beispiel wiederholen. Sie sollten einfach annehmen, daß jedes Beispiel in diesem Buch mit den folgenden vier Zeilen beginnt:

```
#include <stdio.h>
#include <errno.h>
#include <fcntl.h>
#include "defs.h"
```

Die Dateien **errno.h** und **fcntl.h** werden wir erklären, wenn wir in Kapitel 2 beginnen, sie zu benutzen.

Wir werden in unseren Beispielen so weit wie möglich Standard-UNIX-Funktionen benützen. Diese sind im Abschnitt 3 des UNIX-Handbuchs beschrieben. Alle in diesem Buch benutzten Standard-Funktionen sind ferner im Anhang B aufgeführt.

1.11 Portabilität

Wir werden jedes Kapitel mit einer Diskussion über Portabilität beenden. Hier wollen wir die UNIX-Portabilität im allgemeinen etwas beleuchten.

[21] Auf Computern mit einer Wortgröße von 32 Bit operieren diese Makros auf den am weitesten rechts stehenden 16 Bits.

"UNIX-Portabilität" bedeutet zumindest drei Dinge:

1. Portabilität von UNIX selbst von einem Computer auf einen anderen.

2. Portabilität von C-Programmen von einem Computer auf einen anderen, wobei beide die gleiche Version von Unix benutzen.

3. Portabilität von C-Programmen von einer UNIX-Version auf eine andere.

Portabilität von UNIX selbst interessiert uns hier nicht, da dies ein Buch über die Externa des Kerns ist, nicht über seine Interna.[22]
Portabilität von C-Programmen auf Computern betrifft uns wenig, vor allem deswegen, da wir in diesem Buch nur sehr wenig von der Sprache C verwenden. Eine umfassende Diskussion dieser Art von Portabilität findet sich in *The C Programming Language.*

Wir sind an der Portabilität von C-Programmen zwischen verschiedenen *UNIX-Versionen* interessiert, insbesondere an jenen, die viele Systemaufrufe benützen. Dies ist wahrhaftig die dunkle Seite von UNIX, der Teil, über den Sie in Anzeigen und Werbebroschüren nichts lesen. Jede Nichtportabilität ist eine Schande, da die Portabilität von Programmen auf Computern, die unter UNIX laufen, vielleicht die größte Stärke von UNIX ist, zumindest im kommerziellen Bereich.

Es ist einfach, die älteren Systemaufrufe, wie **pipe** oder **write**, portabel zu benützen, da diese von vornherein für UNIX so wesentlich waren, daß alle Versionen in Bezug auf sie kompatibel geblieben sind. Die neueren oder anspruchsvolleren Möglichkeiten, wie FIFO-Dateien oder Bildschirmsteuerung, werfen die Probleme auf. Es genügt nicht zu sagen: "Verwenden Sie diese Möglichkeiten nicht" oder "Wenn Sie diese Dinge verwenden, sind Ihre Programme eventuell nicht portabel". Ernsthafte Anwendungen *müssen* diese Möglichkeiten ausnutzen — deshalb gibt es sie ja! Es geht eben nicht, daß man jemandem, der einen Bildschirmeditor programmiert, erzählt, er möge dies ohne Verwendung der Bildschirmsteuerung tun.

Deshalb ist es unser Ziel, Ihnen am Ende jedes Kapitels einige Ratschläge darüber zu geben, wie man das benützt, was wir Ihnen gezeigt haben, und trotzdem portabel bleibt. Manchmal können wir nur wenig mehr tun als die Grenzen der Portabilität aufzuzeigen: Welche Versionen in der Art und Weise arbeiten, wie wir gesagt haben, und welche nicht. Ab und zu können wir mehr tun: Wir werden spezielle Codiertechniken vorschlagen, die die Probleme vermeiden.

Zum Schluß noch ein paar gute Neuigkeiten. Mit der erwähnenswerten Ausnahme der in System V neuen Systemaufrufe, die sich auf die Prozeßkommunikation beziehen, unterscheiden sich System III und System V in den Schnittstellen der

[22] Leser, die an diesem Thema interessiert sind, werden auf den Artikel "Portability of C Programs and the UNIX System" von S.C. Johnson und D.M. Ritchie in *The Bell System Technical Journal* 57, no. 6, pt. 2 (July-August 1978), 2021–48, verwiesen.

Systemaufrufe nur sehr wenig. Selbst die Terminalsteuerung, die sich traditionellerweise mit jeder Version ändert, ist dieselbe geblieben. Während dieses Buch geschrieben wurde,[23] bewegte sich die Industrie rasch auf System III und System V zu (weg von Version 7), so daß sich die Situation ständig verbessert, ausgenommen für die Benutzer von Berkeley 4.2 BSD, das sich in erschreckender Weise von Version 7, seiner ursprünglichen Grundlage, entfernt hat. Wir werden trotzdem unser Bestes tun, Ihnen dabei zu helfen, Programme von einer AT&T-Version auf 4.2 BSD zu portieren; der umgekehrte Weg ist allerdings ausgeschlossen, falls viele der Berkeley-spezifischen Systemaufrufe benützt werden.

Prozeßkommunikation bleibt ein ärgerlicher Schandfleck. Wir werden in Kapitel 7 eine Methode präsentieren, wie man auf eine effiziente, aber portable Weise Prozeßkommunikation betreiben kann.

[23] aber nicht diese Übersetzung... A.d.Ü.

Kapitel 2: Elementare Ein- und Ausgabeoperationen für Dateien

2.1 Einführung

In diesem Kapitel werden wir Ein- und Ausgabe für gewöhnliche Dateien untersuchen. In Kapitel 3 werden wir mit fortgeschritteneren Systemaufrufen weitermachen. In Kapitel 4 wird Ein- und Ausgabe für Special-Files, in Kapitel 6 für Pipes und in Kapitel 7 für FIFO-Dateien behandelt.

Beginnen wir mit einem einfachen Beispiel, das fünf Systemaufrufe verwendet, mit denen Sie vielleicht bereits vertraut sind: **open**, **creat**, **read**, **write** und **close**. Die folgende Funktion kopiert eine Datei in eine andere (wie das **cp**):

```
#define BUFSIZE 512

void copy(from, to)  /* Datei kopieren (enthaelt einen Fehler) */
    char * from, * to;
{
    int fromfd, tofd, nread;
    char buf[BUFSIZE];

    if ((fromfd = open(from, 0)) == -1)
        syserr(from);
    if ((tofd = creat(to, 0666)) == -1)
        syserr(to);
    while ((nread = read(fromfd, buf, sizeof(buf))) > 0)
        if (write(tofd, buf, nread) != nread)
            syserr("write");
    if (nread == -1)
        syserr("read");
```

```
    if (close(fromfd) == -1 || close(tofd) == -1)
        syserr("close");
}
```

Versuchen Sie, den Fehler in dieser Funktion herauszufinden (in Abschnitt 1.9 findet sich ein Anhaltspunkt). Wenn Sie ihn nicht finden, warten Sie ab: Wir werden ihn in Abschnitt 2.7 zeigen.

Wir wollen hier nur kurz auf diese Funktion eingehen; später wird noch genug Zeit sein, um ins Detail zu gehen. Der Aufruf von **open** öffnet die Eingabedatei zum Lesen (wie durch Null als zweites Argument angezeigt wird) und liefert einen Filedeskriptor zum späteren Gebrauch bei Systemaufrufen zurück. **creat** erzeugt eine neue Datei, falls noch keine existiert, oder verkürzt eine bestehende Datei. In jedem Fall wird die Datei zum Schreiben geöffnet und ein Filedeskriptor zurückgeliefert. Das zweite Argument von **creat** sind die Zugriffsschutz-Bits, die bei der Erzeugung der Datei benutzt werden sollen (oktal 0666 bedeutet Lese- und Schreibrecht für den Besitzer, für die Gruppe und für alle anderen). **read** liest so viele Bytes, wie das dritte Argument angibt, in den Puffer, auf den das zweite Argument zeigt. Zurückgeliefert wird die Zahl gelesener Bytes, bzw. Null, falls das Dateiende erreicht ist. **write** gibt so viele Bytes, wie wieder durch das dritte Argument angegeben, aus dem Puffer aus, der über das zweite Argument gefunden wird. Die Zahl der geschriebenen Bytes wird zurückgeliefert. Falls diese ungleich der Anzahl Bytes ist, die geschrieben werden sollten, betrachten wir dies als einen Fehler. Zum Schluß **close** — hiermit wird der Zugriff auf eine Datei abgeschlossen. Beachten Sie, daß wir der Einfachheit halber alle Fehler (Resultatwerte von -1) als tödlich ansehen (die Funktion **syserr** wurde in Abschnitt 1.9 gezeigt). In den meisten Beispielen in diesem Buch werden wir so verfahren.

Unsere Funktion wäre viel schöner geworden, wenn nicht jeder Systemaufruf in eine **if**-Anweisung gepackt worden wäre, um nachzuprüfen, ob ein fehlerhaftes Resultat geliefert wurde. Es ist jedoch ein Armutszeugnis, fehlerhafte Resultatwerte zu ignorieren. Viele Programmierer übergehen sie bei Funktionen, die "niemals" Fehler zurückliefern, wie **read** oder **close**, doch ist dies falsche Sparsamkeit. Der Autor hat unzählige Stunden damit vergeudet, nach Fehlern zu suchen, die man leicht hätte ausfindig machen können, wenn ein Fehler eines Systemaufrufs abgefangen worden wäre, sobald er aufgetreten ist.

2.2 Filedeskriptoren

Jeder UNIX-Prozeß hat 20[1] Filedeskriptoren zur Verfügung, die von 0 bis 19 durchnumeriert sind. Per Konvention sind die ersten drei bereits aktiv, wenn der Prozeß

[1] Systemparameter. A.d.Ü.

startet. Filedeskriptor 0 ist die *Standard-Eingabe*, Filedeskriptor 1 die *Standard-Ausgabe* und Filedeskriptor 2 die *Diagnose-Ausgabe*. Ein UNIX-Filter würde von der Standard-Eingabe lesen und auf die Standard-Ausgabe schreiben. Für wichtige Nachrichten sollte der Filedeskriptor 2 benützt werden, da alles, was zum Filedeskriptor 1 geschrieben wird, in einer Pipe oder in einer Datei verschwinden könnte und nie wiedergesehen würde.

Jeder dieser Standard-Filedeskriptoren könnte mit einer Datei, einer Pipe, einer FIFO-Datei oder einer Gerätedatei verbunden sein. Deshalb ist es am besten, so zu programmieren, daß man von der Art der Quelle oder des Ziels unabhängig ist, was jedoch nicht immer möglich ist. Ein Bildschirmeditor zum Beispiel würde wahrscheinlich überhaupt nicht funktionieren, wenn die Standard-Ausgabe kein Terminal-Gerät ist.

Die drei Standard-Filedeskriptoren sind zur sofortigen Verwendung in **read**- und **write**-Systemaufrufen vorbereitet. Die anderen 17 Filedeskriptoren sind für Dateien, Pipes und Special-Files verfügbar, die der Prozeß für sich selbst öffnet. Einem Vater-Prozeß ist es möglich, einem Sohn-Prozeß mehr als nur die drei Standard-Filedeskriptoren zu hinterlassen. Dies passiert zwar nicht sehr oft, könnte jedoch in bestimmten Anwendungen nützlich sein, die in mehrere eng gekoppelte Prozesse aufgeteilt sind.

Es gibt fünf Systemaufrufe, die Filedeskriptoren erzeugen. Dieses Kapitel behandelt **creat** und **open**, Kapitel 3 beschreibt **fcntl** und Kapitel 6 untersucht **pipe** und **dup**.

2.3 Der Systemaufruf "creat"

Wir werden jeden Systemaufruf einführen, indem wir einen Funktionskopf zeigen, so, als ob wir ihn in C programmieren wollten. Wir verwenden diese Konvention (wie es auch das UNIX-Handbuch tut), weil sie eine kompakte und syntaktisch zulässige Möglichkeit zur Angabe der mnemotechnisch leicht zu merkenden Namen, der Argumenttypen und des Resultattyps bietet.

Hier ist nun der Funktionskopf von **creat**:

```
int creat(path, perms)  /* Datei erzeugen       */
    char * path;        /* Pfadname             */
    int perms;          /* Zugriffsschutz-Bits */

/* liefert Filedeskriptor oder -1 bei Fehler   */
```

creat verhält sich unterschiedlich, je nachdem, ob die zu erzeugende Datei bereits existiert oder nicht. Falls sie noch nicht existiert, wird eine neue Inode angelegt, und in dem Katalog, in dem die Datei kreiert werden soll, wird ein Verweis auf diese erzeugt. Falls **path** keine /-Zeichen enthält, ist dies der Arbeitskatalog. Die

effektive Benutzernummer oder Gruppennummer des Prozesses, der den Systemaufruf ausführt, muß für diesen Katalog Schreibrecht besitzen. Die Rechte der neuen Datei werden so festgesetzt, wie durch das Argument **perms** angegeben — die niedrigsten neun Bits (Lese-, Schreib- und Ausführungsrecht für den Besitzer, die Gruppe und alle anderen) werden betrachtet.[2] Bits, die allerdings in der *Maske für Dateianlage* (*file mode creation mask*) gesetzt sind, werden gelöscht, wie in Abschnitt 9.4 ausführlicher erörtert wird. Besitzer der Datei wird der Benutzer mit der effektiven Benutzer- und Gruppennummer des ausführenden Prozesses. Weiterhin hat die Datei zu Beginn die Länge 0 und ist zum Schreiben geöffnet. Die Zeitstempel der Datei für letzten Zugriff, letzte Änderung und letzte Statusänderung (Erzeugung) werden auf den aktuellen Zeitpunkt gesetzt.

Wenn die Datei bereits existiert, muß für den Katalog, der auf sie verweist, kein Schreibrecht vorhanden sein, sondern nur Ausführungsrecht (Suchrecht). Jedoch muß der Benutzer mit der effektiven Benutzer- und Gruppennummer Schreibrecht auf die Datei selbst besitzen. Weder die Besitzverhältnisse noch der Zugriffsschutz der Datei werden verändert; lediglich die Länge der Datei wird auf Null verkürzt und die Datei zum Schreiben geöffnet. Die Zeitstempel für Änderung und Statusänderung werden auf den aktuellen Zeitpunkt gesetzt. Das Argument **perms** wird ignoriert.

2.4 Der Systemaufruf "unlink"

```
int unlink(path)        /* Verweis entfernen */
    char * path;        /* Pfadname          */

/* liefert 0 bei Erfolg oder -1 bei Fehler  */
```

Der **unlink**-Systemaufruf entfernt einen Verweis aus einem Katalog, indem er die Verweiszahl in der Inode um eins erniedrigt. Falls die resultierende Verweiszahl 0 ergibt, wirft das Dateisystem die Datei weg. Der ganze durch die Datei belegte Plattenplatz wird zur Wiederverwendung zur Verfügung gestellt (d.h. in die freie Liste eingefügt). Auch die Inode steht wieder zur Verfügung.

Wenn die Verweiszahl Null wird, während ein Prozeß die Datei noch geöffnet hat, so wird das Dateisystem mit der Entfernung der Datei solange warten, bis sie geschlossen wird. Dieses Charakteristikum wird oft beim Erzeugen von temporären Dateien ausgenützt:

```
fd = creat("temp", 0666);
unlink("temp");
```

[2] Tatsächlich werden 12 Bits betrachtet, einschließlich des *set-user-ID*-Bits und des *set-group-ID*-Bits, aber man setzt diese beim Erzeugen einer Datei praktisch nie; stattdessen wird fast immer der Systemaufruf **chmod** verwendet.

Diese Technik besitzt zwei Vorteile: Erstens wird die Datei entfernt, falls der Prozeß aus irgendeinem Grunde verenden sollte. Somit braucht man z.B. kein *Hangup*-Signal abfangen, um sicherzugehen, daß die Datei auch wirklich entfernt wird. Zweitens ist die Gefahr geringer, daß ein zweiter Prozeß zufällig die gleiche temporäre Datei benützt, da der Verweis sofort aus dem Arbeitskatalog entfernt wird, sobald **unlink** abläuft. "Weniger" Gefahr ist jedoch immer noch zu viel; zwischen den Aufrufen von **creat** und **unlink** besteht eine kleine Zeitspanne, in der ein anderer Prozeß hereinschleichen und **creat** auf die gleiche Datei anwenden könnte. Eine sicherere Methode wäre, die Prozeßnummer in den Dateinamen einzubetten (z.B. "temp21763", wobei 21763 die Prozeßnummer darstellt), da die Eindeutigkeit von Prozeßnummern garantiert ist. Der Systemaufruf **getpid** (s. Abschnitt 5.7) kann dazu benützt werden, die Prozeßnummer zu erhalten. Tatsächlich besitzen die meisten UNIX-Systeme sogar eine Funktion namens **tmpnam** in der C-Bücherei, die temporäre Dateinamen erfindet.

Nur der Super-User kann einen Katalog entfernen (siehe Abschnitt 3.2).

2.5 Implementierung von Semaphoren mittels Dateien

Wenn **creat** eine *neue* Datei erzeugt, so wird diese auch dann zum Schreiben geöffnet, wenn das Argument, das die Rechte angibt, Schreiben gar nicht erlaubt. Wenn **creat** jedoch versucht, eine *existierende* Datei zu verkürzen, so mißlingt dies, falls kein Schreibrecht vorhanden ist. Dies erlaubt es, eine Datei als ein Semaphor zu verwenden, da **creat** nur gelingt, falls die Datei noch nicht existiert. Zwei Prozesse, die alleinigen Zugriff auf ein Betriebsmittel haben wollen, können nach folgendem Protokoll vorgehen: Bevor sie auf das Betriebsmittel zugreifen, versuchen sie, eine Datei (die einen vereinbarten Namen besitzt) zu erzeugen, ohne daß diese Datei Schreibzugriff erlaubt. Nur einer der Prozesse wird Erfolg haben; der **creat**-Aufruf des anderen Prozesses wird mißlingen. Dieser kann entweder abwarten und es später nochmal versuchen, oder er gibt einfach auf. Wenn der erfolgreiche Prozeß seine Arbeit am Betriebsmittel beendet hat, entfernt er die Semaphor-Datei. Der **creat**-Aufruf des vorher erfolglosen Prozesses wird nun gelingen, und er kann gefahrlos fortfahren.

Ironischerweise können nur *gewöhnliche* Benutzer Semaphore mittels **creat** implementieren — die Methode würde beim *Super-User* nicht funktionieren. Der Grund dafür liegt darin, daß ein vom Super-User abgesetzter **creat**-Aufruf eine bestehende Datei erfolgreich verkürzt und öffnet, selbst wenn die Datei das Schreiben verbietet (es gibt nichts, was der Super-User nicht darf). Tatsächlich litt der UNIX-Editor **ed** früher einmal an diesem Problem. Der Editor benützte einen Semaphor-Mechanismus, ähnlich dem von uns beschriebenen, um jedem Benutzer eine eindeutige temporäre Datei zuzuordnen. Ab und zu geschah es zufällig, daß

ein Benutzer solch eine temporäre Datei mit dem Super-User teilte, der normalerweise die Paßwortdatei editierte. Das äußerte sich dadurch, daß ein Stück aus der Paßwortdatei mitten in der Datei des Benutzers auftauchte! Im nächsten Abschnitt werden wir zeigen, wie dieses Problem zu lösen ist; für den Moment nehmen wir an, daß nur gewöhnliche Benutzer Semaphore verwenden.

Das Semaphor-Protokoll wird am besten in zwei Funktionen **lock** und **unlock** zusammengefaßt, um wie folgt verwendet zu werden:

```
if (lock("accounts"))
{
    ...Zugriff auf...
    unlock("accounts");
}
else
    ...Zugriff nicht erhalten...
```

Der Sperrname "accounts" ist abstrakt; er braucht mit einer wirklichen Datei überhaupt nichts zu tun zu haben. Wenn zwei oder mehr Prozesse dieses Programmstück gleichzeitig ausführen, wird sie die Sperre davon abhalten, den geschützten Bereich ("...Zugriff auf...", was immer das heißen mag) gleichzeitig auszuführen. Denken Sie aber daran, daß natürlich kein Schutz besteht, falls ein Prozeß **lock** nicht aufruft.

Hier ist der Code für **lock** und **unlock**:

```
#define LOCKDIR "/tmp/"
#define MAXTRIES 3
#define NAPTIME 5

BOOLEAN lock(name)  /* in den Besitz der Sperre gelangen */
    char * name;
{
    char * path, * lockpath();
    int fd, tries;
    extern int errno;

    path = lockpath(name);
    tries = 0;
    while ((fd = creat(path, 0)) == -1 && errno == EACCES)
    {
        if (++tries >= MAXTRIES)
            return(FALSE);
        sleep(NAPTIME);
    }
```

```
    if (fd == -1 || close(fd) == -1)
        syserr("lock");
    return(TRUE);
}

void unlock(name)              /* Sperre freigeben              */
    char * name;
{
    char * lockpath();

    if (unlink(lockpath(name)) == -1)
        syserr("unlock");
}

static char * lockpath(name)  /* Pfad der Sperrdatei erzeugen */
    char * name;
{
    static char path[20];
    char * strcat();

    strcpy(path, LOCKDIR);
    return(strcat(path, name));
}
```

Die Funktion **lockpath** erzeugt einen wirklichen Dateinamen, der als Sperre zu verwenden ist. Wir stecken ihn in den Katalog **/tmp**, da dieser auf jedem UNIX-System existiert und für jedermann schreibbar ist. Um eine Sperre anzubringen, versuchen wir die Datei so zu erzeugen, daß sie keine Rechte besitzt (und damit natürlich auch kein Schreibrecht). Dies kann nur gelingen, falls die Datei noch nicht existiert. Wir sind so vorsichtig, **errno** abzufragen, falls **creat** schiefgeht, da der einzige annehmbare Grund dafür fehlendes Schreibrecht der Datei ist. Die Konstante **EACCES** ist in der Einleitung von Kapitel 2 des UNIX-Handbuchs beschrieben. Symbolische Konstanten für Fehlernummern sind in der Definitionsdatei **/usr/include/errno.h** definiert. Rufen Sie sich aus Abschnitt 1.10 in Erinnerung, daß diese Datei als eingefügt vorausgesetzt wird.

Wir versuchen höchstens **MAXTRIES**-mal, die Datei zu erzeugen, jeweils mit einer Pause von **NAPTIME** Sekunden zwischen den einzelnen Versuchen. Die Funktion **sleep** ist Bestandteil der C-Bücherei; in Abschnitt 8.7 werden wir sehen, wie man sie selbst programmiert. Weiterhin schließen wir den Filedeskriptor, den wir von **creat** bekamen, da wir nicht daran interessiert sind, wirklich etwas in die Datei zu schreiben. Als Verbesserung könnten wir unsere Prozeßnummer und die aktuelle Zeit in die Datei schreiben, so daß andere wartende Prozesse sehen könnten, auf

wen sie warten und wie lange derjenige schon arbeitet. Wenn wir dies wollten, so hätten wir die Rechte als oktal 444 anstelle von 0 angegeben.

Alles, was **unlock** zu tun hat, ist, die Datei zu entfernen. Der nächste Versuch, sie zu erzeugen, wird dann gelingen.

Der Gebrauch einer Datei als Semaphor bringt ein wenig unnötigen Aufwand mit sich, besitzt jedoch den Vorteil, auf jeder UNIX-Version zu funktionieren. Diese Technik wird von vielen Kommandos benutzt, die mit UNIX mitgeliefert werden.

Eine Annahme über **creat** ist entscheidend für die Verwendung bei Semaphoren: **creat** muß atomar sein. Wenn zwei Prozesse gleichzeitig versuchen, die gleiche Datei zu erzeugen, ohne daß diese Schreibzugriff erlaubt, dann wird höchstens einer von ihnen Erfolg haben. Das ist eine wohlbekannte Tatsache, und es gibt zu viele wichtige Kommandos, die darauf vertrauen, als daß dies in einer zukünftigen UNIX-Version nicht mehr gelten könnte. Die Seite im Handbuch von Version 7 über **creat** besagt, daß dieses atomar ist, die Handbücher von System III und System V tun dies jedoch nicht.

Eine andere wohlbekannte, aber undokumentierte Tatsache über **creat** ist, daß der kleinste verfügbare Filedeskriptor zurückgeliefert wird. Normalerweise interessiert uns die wirkliche Zahl nicht. Wenn wir jedoch eine Pipe öffnen, kümmern wir uns sehr wohl darum (siehe Abschnitt 6.4).

2.6 Der Systemaufruf "open"

Es gibt zwei Formen des Systemaufrufs **open**. Die ältere Form, mit zwei Argumenten, wird zum Lesen oder Schreiben existierender Dateien verwendet. Die neuere Form (neu seit System III), mit drei Argumenten, kann alles, was die ältere kann, alles, was **creat** kann, und sogar noch mehr. Wir beginnen mit der älteren Form:

```
int open(path, flags)  /* Datei oeffnen                    */
    char * path;       /* Pfadname                         */
    int flags;         /* Lesen, Schreiben oder beides */

/* liefert einen Filedeskriptor oder -1 bei Fehler      */
```

Die Datei, die durch **path** angegeben wird, muß bereits existieren. Wenn **flags** gleich 0 ist, wird sie zum Lesen geöffnet, bei 1 zum Schreiben und bei 2 sowohl zum Lesen als auch zum Schreiben. Der zurückgelieferte Filedeskriptor kann in späteren Systemaufrufen, wie **lseek**, **read**, **write** und **close**, benützt werden. Die effektive Benutzernummer oder die effektive Gruppennummer müssen, abhängig vom Wert von **flags**, Lese- und/oder Schreibrecht besitzen. Der Positionszeiger wird auf das erste Zeichen der Datei gesetzt. Auch wenn die Datei sowohl zum Lesen als auch zum Schreiben geöffnet wird, gibt es nur einen einzigen Positionszeiger, so daß im

allgemeinen jedem Aufruf von **read** oder **write** ein Aufruf von **lseek** (siehe Abschnitt 2.11) vorausgehen wird.

Da **creat** eine Datei nur zum Schreiben öffnet, wird oft ein kurzes Programmstück wie das nachfolgende verwandt, um eine Datei zu erzeugen und sie zum Lesen und zum Schreiben zu öffnen:

```
if ((fd = creat(path, 0600)) == -1)
    syserr("creat");
if (close(fd) == -1 || (fd = open(path, 2)) == -1)
    syserr("reopen");
```

Beachten Sie, daß dies mißlingt, falls die Datei vor dem **creat**-Aufruf bereits existiert und keinen Lese- oder Schreibzugriff erlaubt, da dann **creat** die Rechte nicht ändern würde. Es wäre besser, wenn man **unlink** vor dem Aufruf von **creat** aufrufen würde:

```
if (unlink(path) == -1)
    syserr("unlink");
```

Nun kommen wir zur raffinierteren Form von **open**:

```
#include <fcntl.h>

int open(path, flags, perms )  /* Datei oeffnen       */
    char * path;               /* Pfadname            */
    int flags;                 /* Optionen-Flaggen    */
    int perms;                 /* Zugriffsrechte-Bits */

/* liefert einen Filedeskriptor oder -1 bei Fehler    */
```

In diesem Fall nimmt das Argument **flags** neben 0, 1 und 2 noch weitere Werte an. Weiterhin gibt es ein drittes Argument namens **perms**, das bei einigen Werten von **flags** die gleiche Bedeutung wie das zweite Argument bei **creat** besitzt. Anstatt für **flags** Zahlen zu verwenden, sind in der Definitionsdatei **/usr/include/fcntl.h** symbolische Konstanten definiert. Diese können mit dem Operator | ge**ODER**t werden, um verschiedene kombinierte Effekte zu erreichen. Allerdings sind nicht alle Kombinationen sinnvoll.

Nachfolgend sind die symbolischen Konstanten aufgeführt. Beachten Sie, daß die ersten drei, die sich gegenseitig ausschließen, die Werte 0, 1 und 2 besitzen, und daher mit der alten Form von **open** kompatibel sind.

O_RDONLY	Öffnen zum Lesen.
O_WRONLY	Öffnen zum Schreiben.
O_RDWR	Öffnen zum Lesen und Schreiben.
O_NDELAY	Wenn Pipes, FIFO-Dateien und Kommunikationsgeräte geöffnet werden, gibt diese Flagge an, ob **open** warten oder sofort zurückkehren soll. Davon sind auch spätere Lese- und Schreiboperationen betroffen. Nähere Einzelheiten hierzu finden Sie in den Kapiteln 4, 6 und 7. **O_NDELAY** hat keinen Einfluß auf gewöhnliche Dateien und Kataloge.
O_CREAT	Falls die Datei nicht existiert, wird sie erzeugt (in der Art von **creat**). Dies ist der einzige Fall, in dem das dritte Argument (**perms**) benützt wird.
O_TRUNC	Falls die Datei existiert, wird sie auf Länge Null verkürzt (selbst wenn sie nicht zum Schreiben geöffnet werden soll).
O_EXCL	Wenn **O_CREAT** ebenfalls gesetzt ist, mißlingt der Aufruf, falls die Datei bereits existiert.
O_APPEND	Alle weiteren Schreiboperationen erfolgen an das Dateiende.
O_WSYNC	Diese Flagge ist in Xenix 3 verfügbar, aber in keiner AT&T UNIX-Version. Bei jedem Schreibzugriff wird solange gewartet, bis die physikalische Ausgabe beendet ist. Wir behandeln diesen Aspekt weiter in Abschnitt 2.7.

Die Flaggen **O_CREAT** und **O_TRUNC** spalten die Fähigkeiten von **creat** auf: Man kann nun aus einem à la carte Menü auswählen, anstatt die komplette Mahlzeit nehmen zu müssen. Der **creat**-Systemaufruf ist also äquivalent zu folgender Kombination:

```
O_WRONLY | O_CREAT | O_TRUNC
```

Ein größerer Vorteil der neuen Form von **open** liegt hauptsächlich in der Erleichterung vieler alltäglicher Arbeiten. Stellen Sie sich zum Beispiel ein Programm vor, das eine Datei bei Existenz öffnet und sie sonst erzeugt. Üblicherweise versucht man, sie zuerst zu öffnen; wenn **open** mißlingt, weil die Datei nicht existiert, wird **creat** aufgerufen:

```
if ((fd = open(path, 1)) == -1)
    if (errno == ENOENT)
    {
        if ((fd = creat(path, 0666)) == -1)
            syserr("creat");
    }
```

```
    else
        syserr("open");
```

(Beachten Sie den Gebrauch der geschweiften Klammern, um zu verhindern, daß das **else** mit dem falschen **if** ein Paar bildet.)

Programmierer machen in dieser Gegend oft Fehler (**errno** nicht korrekt zu prüfen ist der häufigste) und zerstören so einige Dateien, bis die Fehler behoben sind. Der neuere **open**-Aufruf vereinfacht diese alltägliche Aufgabe und hilft somit, Fehler zu vermeiden:

```
if ((fd = open(path, O_WRONLY | O_CREAT, 0666)) == -1)
    syserr("open");
```

O_TRUNC ist die wirklich gefährliche Flagge. Solange sie aus dem Spiel bleibt, kann man eine bestehende Datei mittels **open** nicht zerstören.

Auch das gezeigte Programmfragment in diesem Abschnitt, das eine Datei mit der alten Form erzeugt und zum Lesen und Schreiben öffnet, kann nun viel direkter formuliert werden:

```
if ((fd = open(path, O_RDWR | O_CREAT, 0600)) == -1)
    syserr("open");
```

Man darf mit Recht fragen, warum **creat** immer noch als Systemaufruf in UNIX vorhanden ist. Schließlich wurden andere alte Systemaufrufe wie **sleep** zu Unterprogrammen degradiert, als allgemeinere Möglichkeiten eingeführt wurden (**alarm** und **pause**). Die Antwort ist wahrscheinlich gefühlsmäßig; **creat** ist von Anfang an ein Systemaufruf gewesen, und ihm wurde offensichtlich ein Anspruch darauf zugestanden.

Die Flagge **O_EXCL** ist ein offizieller Weg, eine Datei als Semaphor zu betrachten; es ist nicht länger nötig, den Trick mit der Erzeugung einer Datei ohne Schreibrechte zu verwenden, wie wir es im vorigen Abschnitt taten. Am wichtigsten ist, daß ein Semaphor, das mit **O_EXCL** verwirklicht wird, auch für den Super-User funktioniert. Deshalb können wir in der Funktion **lock** die Zeile:

```
while ((fd = creat(path, 0)) == -1 && errno == EACCES)
```

ersetzen durch:

```
while ((fd = open(path, O_WRONLY | O_CREAT | O_EXCL, 0666)) == -1
        && ERRNO == EEXIST)
```

Es ist frustrierend, daß das UNIX-Handbuch nicht explizit angibt, daß zwei Prozesse, wenn sie gleichzeitig die gleiche Datei mit **open** und mit gesetzten Flaggen **O_EXCL** und **O_CREAT** öffnen wollen, nicht beide erfolgreich sein können (das heißt,

daß **open** atomar ist). Das Handbuch impliziert dies nur, es gilt aber wirklich als Tatsache.

Warum muß **O_EXCL** mit **O_CREAT** gepaart werden? Oder anders: Warum kann man existierende Dateien nicht zum alleinigen Zugriff öffnen? Die Antwort ist die: Nur weil eine UNIX-Version die **O_EXCL**-Flagge besitzt, hat sie noch lange keine separaten unabhängigen Semaphore. Die Existenz der Datei selbst stellt immer noch das Semaphor dar. Könnte eine bestehende Datei zum alleinigen Zugriff geöffnet werden, so müßte etwas anderes als Semaphor herhalten (zum Beispiel eine andere Datei). Deshalb passiert es selten, daß eine Datei, die als Semaphor verwendet wird, in Wirklichkeit irgendwelche Daten des Benutzers enthält. Eine Datei ist entweder ein Semaphor oder ein Datenspeicher, aber nicht beides zusammen.

Die Flagge **O_APPEND** ist für Programme vorgesehen, die etwa ein Logbuch führen oder Buchführungsgeschäfte fortschreiben. Mehrere Prozesse könnten alle an das Ende der gleichen Datei schreiben wollen, um Daten in chronologischer Reihenfolge zu speichern. Sie besitzen jedoch alle ihren eigenen Positionszeiger und daher ihre eigene Vorstellung davon, wo sich das Dateiende befindet. Man könnte denken, die Lösung bestehe darin, vor jeder Schreiboperation das Dateiende aufzusuchen; da **lseek** und **write** jedoch zwei verschiedene Systemaufrufe sind, besteht die Möglichkeit, daß sich in der dazwischenliegenden Zeitspanne ein anderer Prozeß hereinschleicht und schreibt. Dies hätte zur Folge, daß der erste Prozeß die Daten des zweiten Prozesses überschreibt. Um dies zu verhindern, können wir unseren Semaphor-Mechanismus einsetzen:

```
if (!lock("accts"))
    syserr("lock");
if (lseek(fd, 0L, 2) == -1)  /* ans Ende positionieren */
    syserr("lseek");         /* siehe Abschn. 2.11     */
if (write(fd, acctrec, sizeof(acctrec)) == -1)
    syserr("write");
unlock("accts");
```

Dies funktioniert zwar, aber mit enormen Kosten infolge der Implementierung von **lock**. Wenn wir stattdessen die Datei mit der Flaggenkombination

```
O_WRONLY | O_CREAT | O_APPEND
```

öffnen, so geht jeder Schreiboperation automatisch ein Positionieren an das Dateiende voraus, und die Aufrufe von **lock**, **lseek** und **unlock** sind unnötig.

Wiederum versäumt es das UNIX-Handbuch, festzustellen, daß Schreibzugriffe auf Dateien, die mit **O_APPEND** geöffnet wurden, atomar sind. Für unbedenkliche Daten genügt diese Vermutung (sie könnte sogar wahr sein); für kritische Daten sollte, um sicherzugehen, ein Semaphor verwendet werden. Denken Sie jedoch

daran, daß auch unsere Implementierung von **lock** auf solchen nichtdokumentierten Annahmen aufgebaut ist.

Die Gefahr solcher Annahmen liegt nicht darin, daß sich eine spezielle UNIX-Implementierung unberechenbar verhalten könnte — wir können immer experimentieren oder den Verkäufer fragen, um die Tatsachen herauszubekommen. Das Problem liegt darin, daß unsere Programme nicht unbedingt auf eine andere UNIX-Version übertragbar sein könnten, da Implementierer nur an das gebunden sind, was im Handbuch steht.[3] Und noch schlimmer: Selbst ein System, das von **open** und **write** keine Unteilbarkeit fordert, wird sich in 99 Prozent aller Fälle trotzdem so verhalten und macht dadurch das Erkennen des Fehlers äußerst schwer.

2.7 Der Systemaufruf "write"

```
int write(fd, buf, nbytes)   /* in Datei schreiben              */
    int fd;                  /* Filedeskriptor                  */
    char * buf;              /* Pufferadresse                   */
    unsigned nbytes;         /* Zahl zu schreibender Zeichen    */

/* liefert die Zahl geschriebener Zeichen oder -1 bei Fehler */
```

write schreibt die **nbytes** Zeichen, auf die **buf** zeigt, in die geöffnete Datei, die durch **fd** angegeben ist. Der Schreibvorgang beginnt bei der aktuellen Position des Positionszeigers, und nach dem Schreiben wird der Positionszeiger um die Zahl der gelesenen Bytes weitergesetzt. Zurückgeliefert wird die Zahl der geschriebenen Zeichen oder -1, falls ein Fehler auftrat.

write wird auch dazu benützt, in Pipes und Special-Files zu schreiben, doch ist die Bedeutung in diesen Fällen etwas verschieden. Wir verschieben die Erörterung von Schreibzugriffen auf andere als gewöhnliche Dateien auf die Kapitel 4, 6 und 7.

write ist trügerisch einfach. Man ist ermutigt, zu meinen, daß **write** die Daten schreibt und dann zurückkehrt, doch ein wenig Herumprobieren wird davon überzeugen, daß dies unmöglich ist. Zum Beispiel ließ ich auf meinem eigenen UNIX-System ein Programm laufen, das 20 Blöcke zu je 512 Bytes schrieb, und das vor jedem Schreibzugriff an eine andere Stelle in der Datei positionierte. Dies benötigte insgesamt 0.3 Sekunden effektive Zeit. Meine Platte hat eine mittlere Zugriffszeit von ungefähr 0.075 Sekunden, so daß die verbrauchte effektive Zeit nicht einmal für 20 Positioniervorgänge ausreicht, geschweige denn für die Zeit zur

[3] Die Standard-Bemühungen von */usr/group* zielen vermutlich darauf ab, wichtige Eigentümlichkeiten wie die Unteilbarkeit von **creat**, **open** und **write** zu dokumentieren. Ihr vorgeschlagener Standard tut dies jedoch nicht. Wenn dieses Versäumnis absichtlich ist — d.h., eine angepaßte Implementierung braucht *nicht* zu garantieren, daß diese Systemaufrufe atomar sind — dann ist der Standard ernsthaft unvollständig.

Datenübertragung und die anderen im Kern notwendig stattfindenden Vorgänge. Offensichtlich hat jemand geschummelt.

Tatsächlich mogelt das Dateisystem. Wenn Sie einen **write**-Systemaufruf absetzen, so führt dieser nicht einfach den Schreibzugriff aus und kehrt dann zurück. Er überträgt die Daten nur an das Blockdepot, kehrt zurück und behauptet nicht mehr als folgendes:

> *Ich habe Ihre Anfrage zur Kenntnis genommen und habe, in der Annahme, daß Ihr Filedeskriptor in Ordnung ist, und es noch genug Platz auf der Platte gibt, Ihre Daten erfolgreich kopiert. Später, wenn es mir gerade paßt, und wenn ich noch am Leben bin, versuche ich die Daten dorthin auf die Platte zu schreiben, wo sie hingehören. Entdecke ich einen Fehler, werde ich versuchen, etwas auf die Konsole zu schreiben, werde Ihnen jedoch nichts darüber mitteilen (tatsächlich könnten Sie inzwischen aufgehört haben zu existieren). Falls Sie oder irgendein anderer Prozeß versuchen, diese Daten zu lesen, bevor ich sie (auf die Platte) geschrieben habe, so hole ich sie für Sie aus dem Blockdepot. Falls also alles gutgeht, werden Sie niemals herausbekommen, wann und ob ich Ihre Anfrage abgeschlossen habe. Weitere Fragen dürfen Sie nicht stellen. Vertrauen Sie mir einfach. Und danken Sie mir für die schnelle Antwort.*

Wenn alles gutgeht, ist verzögertes Schreiben phantastisch. Die Bedeutung ist dieselbe, wie wenn das Schreiben tatsächlich stattgefunden hätte, es geht jedoch viel schneller. Falls es jedoch einen Plattenfehler gibt, oder der Kern aus irgendeinem Grund anhält, dann ist das Spiel aus. Wir entdecken dann, daß die Daten, die wir "schrieben", überhaupt nicht auf der Platte sind.

Zusätzlich zur Unsicherheit, wann genau der physikalische Schreibvorgang stattfindet, gibt es bei verzögerten Schreiboperationen noch zwei weitere Probleme: Erstens kann ein Prozeß, der eine Schreiboperation beginnt, nicht über Schreibfehler informiert werden. Tatsächlich gehören die Puffer des Dateisystems nicht einzelnen Prozessen; schreiben mehrere Prozesse gleichzeitig in den gleichen Block derselben Datei, so werden ihre Daten in den gleichen Puffer übertragen. Natürlich könnte man sich ein Verfahren vorstellen, bei dem jedem Prozeß, der in einen bestimmten Puffer geschrieben hat, ein "Schreibfehler"-Signal geschickt wird, doch was sollte ein Prozeß zu diesem späten Zeitpunkt damit anfangen? Und wie benachrichtigt der Kern Prozesse, die bereits beendet sind?

Das zweite Problem ist, daß die *Reihenfolge* der physikalischen Schreibvorgänge nicht kontrolliert werden kann. Die Reihenfolge ist aber oft von Bedeutung. Es ist zum Beispiel besser, beim Schreiben einer verketteten Liste in eine Datei zuerst den neuen Satz zu schreiben und dann den Zeiger darauf zu verändern als umgekehrt. Dies deshalb, weil ein Satz, auf den nicht gezeigt wird, im allgemeinen

ein kleineres Problem darstellt als ein Zeiger, der nirgends hinzeigt. Doch selbst wenn die **write**-Systemaufrufe in einer bestimmten Reihenfolge abgesetzt werden, bedeutet dies noch nicht, daß die Puffer auch in dieser Reihenfolge physikalisch auf die Platte geschrieben werden. Deshalb sind vorsichtige Austauschtechniken, von denen dies nur ein Beispiel ist, nicht so vorteilhaft, wie sie sein könnten. Sie bieten Schutz, wenn ein Prozeß zu einem ungünstigen Zeitpunkt verendet, nicht jedoch vor Plattenfehlern oder Abstürzen des Kerns.[4]

Wir wir im vorigen Abschnitt erwähnten, besitzt Xenix 3 eine Flagge für **open** (**O_WSYNC**), die den **write**-Systemaufruf dazu zwingt, seine Arbeit vollständig zu erledigen, bevor er zurückkehrt. Dies verlangsamt das gesamte System merklich, lohnt sich jedoch in kritischen Fällen. Benutzer, die nicht unter Xenix arbeiten, müssen sich darauf verlegen, *nackte* Gerätedateien (*raw*) zu schreiben, wie in Abschnitt 3.3 beschrieben, was allerdings ein hoher Preis ist.

Diese Probleme bei **write** sollten nicht überbetont werden. Betrachtet man, wie zuverlässig Computer heutzutage sind, und wie zuverlässig UNIX-Implementierungen gewöhnlich sind, so sind Abstürze des Kerns ziemlich selten. Die meisten Benutzer sind über die schnellen Antwortzeiten erfreut, die durch das Blockdepot erzielt werden, und finden niemals heraus, daß der Kern mogelt.

Betrachten wir nun nochmals das Beispiel vom Kopieren einer Datei vom Anfang dieses Kapitels. Der Fehler befindet sich im Prüfen eines Schreibfehlers:

```
if (write(tofd, buf, nread) != nread)
    syserr("write");
```

Das UNIX-Handbuch behauptet, daß es kein Fehler ist, wenn die von **write** zurückgelieferte Zahl geringer ist als die geforderte. Speziell, wenn die Dateigröße ihre Grenze erreicht hat, wird der nächste Aufruf von **write** diesen Fehler hervorrufen.[5] Das heißt, der Fehler liegt darin, daß der Aufruf von **syserr** einen sinnlosen Wert für **errno** ausgeben wird. Wir könnten die Funktion neu schreiben, um teilweises Schreiben abzufangen und bei einem wirklichen Fehler aufzuhören:

[4] Neuere UNIX-Versionen benützen intern vorsichtige Austauschtechniken, wenn sie Puffer entleeren, die Kataloge oder Daten der freien Liste enthalten. Für Benutzer-Datenstrukturen ist jedoch keine derartige Kontrolle möglich.

[5] In einigen Versionen von System III, und vielleicht auch noch in anderen Versionen, hat das Handbuch unrecht. Falls nur teilweise geschrieben wird, so wird anstelle der entsprechenden Zahl -1 zurückgeliefert. Deshalb könnte unser Programm schließlich doch keinen Fehler gehabt haben.

```
#define BUFSIZE 512

void copy2(from, to)  /* Datei kopieren (zweiter Versuch) */
    char * from, * to;
{
    int fromfd, tofd, nread, nwrite, n;
    char buf[BUFSIZE];

    if ((fromfd = open(from, 0)) == -1)
        syserr(from);
    if ((tofd = creat(to, 0666)) == -1)
        syserr(to);
    while ((nread = read(fromfd, buf, sizeof(buf))) != 0)
    {
        if (nread == -1)
            syserr("read");
        nwrite = 0;
        do
        {
            if ((n = write(tofd, &buf[nwrite], nread - nwrite)) == -1)
                syserr("write");
            nwrite += n;
        }
        while (nwrite < nread);
    }
    if (close(fromfd) == -1 || close(tofd) == -1)
        syserr("close");
}
```

Realistisch betrachtet, scheint dies zu große Umstände zu machen. Vielleicht ergibt diese einfache Lösung mehr Sinn:

```
if ((n = write(tofd, buf, nread)) != nread)
{
    if (n != -1)
        errno = EFBIG;
    syserr("write");
}
```

Die Konstante **EFBIG** ist in **/usr/include/errno.h** definiert, wird jedoch von **write** nicht benützt. Deshalb ist es erlaubt, sie zu stehlen.

2.8 Der Systemaufruf "read"

```
int read(fd, buf, nbytes)  /* aus Datei lesen              */
    int fd;                /* Filedeskriptor               */
    char * buf;            /* Pufferadresse                */
    unsigned nbytes;       /* Zahl der zu lesenden Zeichen */

/* liefert die Zahl gelesener Zeichen,                      */
/* 0 bei EOF oder -1 bei Fehler                             */
```

Der Systemaufruf **read** stellt das Gegenteil von **write** dar. Er liest **nbytes** Zeichen aus der durch **fd** repräsentierten, geöffneten Datei in den Puffer, auf den **buf** zeigt. Der Lesevorgang beginnt bei der augenblicklichen Position des Positionszeigers; nach dem Lesen wird der Positionszeiger um die Anzahl der gelesenen Bytes weitergesetzt. **read** liefert die Zahl gelesener Zeichen, 0, falls das Dateiende erreicht wurde, oder -1 im Falle eines Fehlers.

Im Gegensatz zu **write** kann der **read**-Systemaufruf nicht gut mogeln, indem er die Daten herüberreicht und sie dann erst später liest. Wenn die Daten nicht bereits in einem Puffer des Kerns sind (aufgrund eines vorangegangenen **write**- oder **read**-Aufrufs), muß der Prozeß einfach auf den Kern warten, um sie von der Platte zu bekommen. Normalerweise versucht der Kern die Sache zu beschleunigen, indem er Zugriffsmuster berücksichtigt, die unterstellen, daß aufeinanderfolgende Plattenblöcke nacheinander gelesen werden, und somit vorausliest, um den Bedürfnissen des Prozesses zuvorzukommen. Wenn das System ausreichend schwach belastet ist, so daß Daten für eine Weile in Puffern gehalten werden können, und wenn die Lesezugriffe hintereinander erfolgen, dann ist Vorauslesen durchaus wirkungsvoll.

2.9 Der Systemaufruf "close"

```
int close(fd)          /* Datei schliessen */
    int fd;            /* Filedeskriptor   */

/* liefert 0 bei Erfolg oder -1 bei Fehler  */
```

Das Allerwichtigste, was Sie über den Systemaufruf **close** wissen müssen, ist, daß er praktisch gar nichts tut. Er entleert keine Puffer des Kerns; er sorgt gerade dafür, daß der Filedeskriptor wieder verwendbar wird. Tatsächlich besteht überhaupt keine Notwendigkeit, **close** aufzurufen, wenn der Filedeskriptor nicht wieder benötigt wird (obwohl es besser ist, da irgendjemand eines schönen Tages Ihr Programm ändern könnte). Die Datei wird sowieso geschlossen, wenn der Prozeß zu Ende geht.

2.10 Gepufferte Ein- und Ausgabe

Die Pufferung des Kerns, die wir bis jetzt beschrieben haben, nimmt Lesezugriffe vorweg, verzögert Schreibzugriffe und hält Daten, auf die häufig zugegriffen wird, in einem schnellen internen Speicher. Dieser Abschnitt beschäftigt sich mit einer ganz anderen Art von Pufferung, die von Benutzerprozessen selbst ganz ohne das Wissen des Kerns vorgenommen wird. Wir nennen sie *Benutzerpufferung*, im Gegensatz zur *Kernpufferung*.

Rufen Sie sich aus Kapitel 1 in Erinnerung, daß das UNIX-Dateisystem auf einer Block-Gerätedatei aufgebaut ist, und daher alle Ein- und Ausgabeoperationen des Kerns und alle Kernpufferungen in Einheiten der Blockgröße erfolgen. Für die meisten UNIX-Versionen sind dies 512 Bytes, doch seit kurzem erlauben System V und 4.2 BSD größere Werte, wie 1024 oder 2048. Wir nehmen in unseren Beispielen an, daß die Blockgröße 512 Bytes beträgt, doch sind wir vorsichtig und benützen eine symbolische Konstante (**BUFSIZE**) anstelle einer wirklichen Zahl, um Änderungen zu erleichtern.

Lese- und Schreiboperationen in Stücken von Vielfachen von 512 Bytes, die in einer 512-Bytes-Umgebung vorkommen, sind viel viel schneller als jede andere kleinere Einheit. Um dies zu demonstrieren, übersetzten wir **copy2** aus dem vorigen Abschnitt mit einer Benutzerpuffergröße von 1, indem wir folgende Änderung vornahmen:

```
#define BUFSIZE 1
```

Dann stoppten wir die Laufzeit der beiden Versionen von **copy**, angewandt auf eine 4000 Bytes große Datei, und erhielten diese Ergebnisse (Zeitangaben in Sekunden):

Methode	*Real-Time*	*User-Time*	*System-Time*
512-Byte-Puffer	1.10	0.02	0.23
1-Byte-Puffer	44.00	1.38	41.37

Die *Real-Time* ist die im ganzen verstrichene Zeit (manchmal auch *Zeit-auf-der-Uhr-an-der-Wand* genannt). Die *User-Time* ist die Zeit, die gebraucht wurde, um Anweisungen im Benutzerprozeß auszuführen, und die *System-Time* ist die Zeit, in der der Kern Anweisungen im Auftrag des Prozesses ausgeführt hat.

Die Leistungseinbuße bei Ein- und Ausgabe für gewöhnliche Dateien in Einheiten mit weniger als 512 Bytes ist derart drastisch, daß es einfach nie gemacht wird, außer das Programm ist nur zum gelegentlichen und unregelmäßigen Gebrauch vorgesehen oder die Umstände sind ziemlich ungewöhnlich (z.B. Rückwärtslesen einer

Datei, siehe nächster Abschnitt). Wenn der Algorithmus sehr einfach programmiert ist und kleinere Stücke benutzt, so werden die Daten zwischen dem Kernbereich und dem Benutzerbereich in Blöcken zu 512 Bytes transportiert, und im Benutzerbereich werden dann kleinere Stücke bewegt, wobei gewöhnliche Datenverschiebebefehle benützt werden, keine Systemaufrufe. Das bedeutet, daß Benutzerpufferung anstelle von Kernpufferung stattfindet.

Es ist üblich, eine Reihe von Unterprogrammen zu verwenden, die Lese- und Schreiboperationen, Positionierbewegungen etc. in Einheiten vornehmen, die vom Aufrufer beliebig bestimmt werden können. Diese Unterprogramme erledigen das Puffern automatisch und weichen niemals vom Blockmodell ab. Ein außergewöhnlich schönes Beispiel eines solchen Pakets ist die sogenannte "Standard-Bücherei der Ein- und Ausgabe", die in den meisten Büchern über C und im Abschnitt 3 des UNIX-Handbuchs beschrieben ist.[6]

Um die Grundregeln eines Benutzerpufferung-Pakets zu zeigen, werden wir hier eine vereinfachte Version, genannt **STREAM**, zeigen. Sie unterstützt Lese- und Schreibzugriffe in Einheiten eines einzelnen Zeichens, jedoch keine Positionierbewegungen. Als erstes zeigen wir die Definitionsdatei **stream.h**, die die Benutzer des Pakets einfügen müssen:

```
#define SENOMEM 1001
#define SEINVAL 1002
#define BUFSIZE 512

typedef struct
{
    int fd;              /* Filedeskriptor            */
    char dir;            /* Art: r oder w             */
    int total;           /* Gesamtzahl Zeichen in buf */
    int next;            /* naechstes Zeichen in buf  */
    char buf[BUFSIZE];   /* Puffer                    */
} STREAM;

STREAM * Sopen();
int Sgetc();
BOOLEAN Sputc();
BOOLEAN Sclose();
```

Nun zur Implementierung des Pakets (**stream.c**). Vergessen Sie nicht, daß diese Datei mit den vier **#include**-Anweisungen beginnt, die in Abschnitt 1.10 gezeigt wurden.

[6] Kapitel 7 von *The C Programming Language.*

```
#include "stream.h"

extern int errno;

STREAM * Sopen(path, dir)  /* oeffnen */
    char * path, * dir;
{
    STREAM * z;
    int fd, flags;
    char * malloc();

    switch (dir[0])
    {
        case 'r':
            flags = O_RDONLY;
            break;
        case 'w':
            flags = O_WRONLY | O_CREAT | O_TRUNC;
            break;
        default:
            errno = SEINVAL;
            return(NULL);
    }
    if ((fd = open(path, flags, 0666)) == -1)
        return(NULL);
    if ((z = (STREAM *) malloc(sizeof(STREAM))) == NULL)
    {
        errno = SENOMEM;
        return(NULL);
    }
    z->fd = fd;
    z->dir = dir[0];
    z->total = z->next = 0;
    return(z);
}
```

```
static BOOLEAN readbuf(z)  /* Puffer fuellen */
    STREAM * z;
{
    switch (z->total = read(z->fd, z->buf, sizeof(z->buf)))
    {
        case -1:
            return(FALSE);
        case 0:
            errno = 0;
            return(FALSE);
        default:
            z->next = 0;
            return(TRUE);
    }
}

static BOOLEAN writebuf(z)  /* Puffer entleeren */
    STREAM * z;
{
    int n, total;

    total = 0;
    while (total < z->next)
    {
        if ((n = write(z->fd, &z->buf[total], z->next - total)) == -1)
            return(FALSE);
        total += n;
    }
    z->next = 0;
    return(TRUE);
}

int Sgetc(z)  /* Zeichen holen */
    STREAM * z;
{
    int c;

    if (z->next >= z->total && !readbuf(z))
        return(-1);
    return(z->buf[z->next++] & 0377);
}
```

```
BOOLEAN Sputc(z, c)  /* Zeichen ausgeben */
    STREAM * z;
    char c;
{
    z->buf[z->next++] = c;
    if (z->next >= sizeof(z->buf))
        return(writebuf(z));
    return(TRUE);
}

BOOLEAN Sclose(z)  /* schliessen */
    STREAM * z;
{
    int fd;

    if (z->dir == 'w' && !writebuf(z))
        return(FALSE);
    fd = z->fd;
    free(z);
    return(close(fd) != -1);
}
```

Schließlich modifizieren wir unsere Dateikopierfunktion für die Verwendung des neuen Pakets:

```
#include "stream.h"

void copy3(from, to)    /* Datei kopieren           */
    char * from, * to;  /* (verwendet STREAM-Paket) */
{
    STREAM * stfrom, * stto;
    int c;
    extern int errno;

    if ((stfrom = Sopen(from, "r")) == NULL)
        syserr(from);
    if ((stto = Sopen(to, "w")) == NULL)
        syserr(to);
    while ((c = Sgetc(stfrom)) != -1)
        if (!Sputc(stto, c))
            syserr("Sputc");
    if (errno != 0)
        syserr("Sgetc");
```

```
    if (!Sclose(stfrom) || !Sclose(stto))
        syserr("Sclose");
}
```

Beachten Sie die sorgfältige Berücksichtigung von Fehlerprüfungen und Fehlermeldungen in diesem Paket. Um den Fehlertyp anzugeben, wird **errno** benützt, genauso wie es die Systemaufrufe tun. Es wurden zwei neue Fehlertypen definiert: **SENOMEM** (nicht genügend Speicherplatz vorhanden; ähnlich zu **SENOMEM** vom Kern) und **SEINVAL** (ungültiges Argument; ähnlich zu **EINVAL**). Somit kann der Benutzer von **STREAM** einfach **errno** prüfen, wenn ein Aufruf einen Fehler liefert. Manchmal wird **errno** dann einen Fehler-Code des Kerns enthalten und manchmal einen von **STREAM**. **Sgetc** setzt **errno** bei Dateiende auf 0, zur Unterscheidung von -1 bei fehlerhafter Rückkehr (etwas, was die Standard-Bücherei der Ein- und Ausgabe nicht macht).

Es ist bedauerlich, daß es drei verschiedene Fehlerresultate gibt: **NULL**, **FALSE** und -1. Dies ist bei UNIX-Unterprogrammen allgemein üblich, und C-Programmierer gewöhnen sich schnell daran.[7]

Sie werden eine starke Ähnlichkeit zwischen **STREAM** und einer Untermenge der Standard-Bücherei der Ein- und Ausgabe feststellen. Hier ist eine Version von **copy**, die das Standard-Paket der Ein- und Ausgabe benützt:

```
void copy4(from, to)    /* Datei kopieren            */
    char * from, * to;  /* (verwendet Standard-Paket */
{                       /*  der Ein- und Ausgabe)    */
    FILE * stfrom, * stto;
    int c;

    if ((stfrom = fopen(from, "r")) == NULL)
        syserr(from);
    if ((stto = fopen(to, "w")) == NULL)
        syserr(to);
    while ((c = getc(stfrom)) != EOF)
        putc(c, stto);
    if (fclose(stfrom) == EOF || fclose(stto) == EOF)
        syserr("Sclose");
}
```

Damit Sie den großen Vorteil der Benutzerpufferung verstehen, zeigt die folgende Tabelle die in Sekunden angegebenen Zeiten für das Kopieren, wenn man einerseits

[7] Obwohl **NULL** und **FALSE** die gleiche Definition (0) besitzen, sind sie unterschiedliche Symbole, da sie verschiedenen Zwecken dienen. **NULL** ist ein ungültiger Zeiger, während **FALSE** ein Boole'scher Wert ist.

STREAM und andererseits die Standard-Bücherei der Ein- und Ausgabe verwendet. (Der Zweckmäßigkeit halber haben wir die Zeiten, die wir mittels direkten Systemaufrufen bekamen, wiederholt.)

Methode	*Real-Time*	*User-Time*	*System-Time*
512-Byte-Puffer	1.10	0.02	0.23
1-Byte-Puffer	44.00	1.38	41.37
STREAM-Paket	1.63	1.33	0.18
Standard-Paket der Ein- und Ausgabe	1.43	1.08	0.20

Mit der Benutzerpufferung haben wir fast das beste aus zwei Welten erhalten: Wir können die Daten verarbeiten wie wir wollen, sogar Byte für Byte, und doch erhalten wir einen Wert für *System-Time* wie ungefähr auch bei der 512-Bytes Puffermethode, mit einem Wert für *User-Time*, der auch nicht schlechter ist als bei der 1-Byte Puffermethode. Somit ist Benutzerpufferung zweifellos der richtige Weg.

Die beinahe komplette Akzeptanz der Standard-Bücherei der Ein- und Ausgabe entbehrt nicht der Ironie. Die Fähigkeit, Ein- und Ausgabe für Dateien in willkürlichen Einheiten vorzunehmen, ist schon immer eines der wirklich erwähnenswerten Merkmale des UNIX-Kerns gewesen. Doch in Wirklichkeit wird diese Fähigkeit fast niemals benützt, da sie zu ineffizient ist.

2.11 Der Systemaufruf "lseek"

```
long lseek(fd, offset, interp)  /* Positionszeiger verschieben */
    int fd;                     /* Filedeskriptor              */
    long offset;                /* Offset in Datei             */
    int interp;                 /* Interpretation von offset   */

/* liefert einen Positionszeiger oder -1 bei Fehler           */
```

Der Systemaufruf **lseek** setzt einfach den Positionszeiger für den nächsten **read**- oder **write**-Aufruf. Es wird keine eigentliche Ein- oder Ausgabe vorgenommen, und es werden auch keine Befehle an den Platten-Controller geschickt. Falls **interp** Null ist, so stellt **offset** direkt eine Position in die Datei dar (das erste Byte der Datei besitzt den Offset 0). Falls **interp** 1 ist, so wird der Positionszeiger um **offset** inkrementiert (**offset** darf auch negativ sein), und falls **interp** 2 ist, so wird der Positionszeiger auf die Größe der Datei plus **offset** gesetzt. Die letzte Möglichkeit wird normalerweise

dazu benutzt, um vom Dateiende her rückwärts zu positionieren, so daß **offset** üblicherweise negativ oder gleich Null ist, wenn **interp** 2 ist.

Das zweite Argument darf eigentlich jeden beliebigen Wert besitzen, solange der resultierende Positionszeiger nicht negativ wird. Der Positionszeiger darf auch hinter das Dateiende zeigen. In diesem Falle dehnt der nächste **write**-Aufruf die Datei auf die nötige Länge, indem er die Lücke mit irgendwelchen unsinnigen Zeichen füllt. Andererseits kehrt ein **read**-Aufruf mit Null (Dateiende) zurück, wenn der Positionszeiger an oder hinter das Dateiende positioniert ist. Wird ein **read**-Aufruf auf die Lücke angewandt, die durch einen **write**-Aufruf hinter das Dateiende erzeugt wurde, so gelingt dieser zwar, doch sind die gelieferten Daten bedeutungslos.

Wenn ein **write**-Aufruf hinter das Dateiende greift, so werden in den meisten UNIX-Systemen die dazwischenliegenden Blöcke, die irgendwelchen Unsinn enthalten, nicht wirklich gespeichert. Deshalb ist es möglich, daß eine Platte mit zum Beispiel 30000 verfügbaren Blöcken Dateien enthält, deren gesamte Länge mehr als diese 30000 Blöcke ausmacht. Dies kann zu ernsten Problemen führen, wenn diese Dateien auf Band gesichert und später restauriert werden. Der Grund liegt darin, daß die Dateien gelesen werden müssen, um sie auf Band zu übertragen, und deshalb mehr als 30000 Blöcke auf dem Band sein werden! Programmierer, die viele solche Dateien mit Löchern erzeugen, werden gewöhnlich von ihrem Systemadministrator hören.

Von allen möglichen Arten, **lseek** zu benützen, gibt es drei, die am beliebtesten sind. Erstens kann **lseek** dazu verwandt werden, eine absolute Position in der Datei aufzusuchen:

```
if (lseek(fd, offset, 0) == -1L)
    syserr("lseek");
```

Zweitens kann man mit Hilfe von **lseek** an das Dateiende gelangen:

```
if (lseek(fd, 0L, 2) == -1L)
    syserr("lseek");
```

Drittens kann man mittels **lseek** die aktuelle Position des Positionszeigers herausfinden:

```
long where, lseek();

if ((where = lseek(fd, 0L, 1)) == -1L)
    syserr("lseek");
```

Die anderen Verwendungsarten von **lseek** sind wesentlich seltener. Einige warnende Worte zu **lseek**: Vergessen Sie nicht, **lseek** in Ihrem Programm zu deklarieren, da sonst der Resultatwert falsch interpretiert wird. Dazu genügt:

```
long lseek();
```

Weiterhin sollten Sie nicht vergessen, daß das zweite Argument **long** sein muß. Beachten Sie den Gebrauch der long-Konstanten **0L** anstatt einer schlichten 0. Denken Sie daran, daß der C-Compiler Funktionsargumente weder prüft noch umwandelt. Er wandelt Resultatwerte um, jedoch nur, wenn Sie die Funktion richtig deklarieren.

Die meisten durch den Kern vorgenommenen Positionierbewegungen sind eher implizit als das Ergebnis expliziter Aufrufe von **lseek**. Wenn **open** oder **creat** aufgerufen wird, so positioniert der Kern auf das erste Byte. Wird **read** oder **write** aufgerufen, so inkrementiert der Kern den Positionszeiger um die Zahl der gelesenen bzw. geschriebenen Bytes. Wird eine Datei mit der **O_APPEND**-Flagge geöffnet, so geht jedem Schreibzugriff ein Positionieren an das Dateiende voraus.

Der Name **lseek** ist fremdartig. Wofür steht das "l"? In den frühen Tagen von UNIX hieß der Systemaufruf **seek**. Das war in der Zeit, als es in C noch keine long-Integer gab. Bei **seek** war das zweite Argument eine Integer (nur 16 Bits; UNIX war damals noch nicht portabel), und es gab *sechs* verschiedene Werte für das dritte Argument. Bei drei von ihnen wurde der Offset als Blocknummer (Vielfaches von 512 Bytes) interpretiert, und bei den anderen drei war der Offset in Bytes ausgedrückt. Um zum Beispiel an die Stelle 1726 zu gelangen, verfuhr man so:

```
if (seek(fd, 3, 3) == -1 || seek( fd, 190, 1) == -1)
    syserr("seek");
```

Der erste **seek**-Aufruf positionierte auf den Anfang des vierten Blocks (Position 1536), der zweite 190 Bytes weiter. Als dann in C long-Integer dazukamen, wurde diese unhandliche Methode überflüssig, da nun Positionszeiger direkt gesetzt werden konnten. Die neue Funktion wurde **lseek** genannt, um Verwirrung zu vermeiden.[8] In der Tat gab es eine Zeitlang ein Unterprogramm namens **seek**, das intern **lseek** verwendete.

Um die Verwendung von **lseek** zu veranschaulichen, zeigen wir Ihnen hier die Funktion **backward**, die die Zeilen einer Datei in umgekehrter Reihenfolge ausgibt. Wenn die Datei etwa folgenden Inhalt besitzt:

```
dog
bites
man
```

so wird **backward** folgende Ausgabe erzeugen:

```
man
bites
dog
```

[8] Der zusätzliche Buchstabe stand zur Verfügung, da **creat** um einen Buchstaben zu kurz war.

Hier ist der Code dafür:

```
void backward(path)  /* Datei rueckwaerts ausgeben */
    char * path;
{
    char s[101], c;
    int i, fd, nread;
    long where, lseek();

    if ((fd = open(path, 0)) == -1)
        syserr(path);
    if ((where = lseek(fd, -1L, 2)) == -1)
        syserr("lseek");
    i = sizeof(s) - 1;
    s[i] = '\0';
    while ((nread = read(fd, &c, 1)) == 1)
    {
        if (c == '\n')
        {
            printf("%s", &s[i]);
            i = sizeof(s) - 1;
        }
        if (i == 0)
            fatal("line too long");
        s[--i] = c;
        if (where == 0)
            break;
        if ((where = lseek(fd, -2L, 1)) == -1)
            syserr("lseek");
    }
    switch (nread)
    {
        case 0:
            fatal("end-of-file");
        case -1:
            syserr("read");
    }
    printf("%s", &s[i]);
    if (close(fd) == -1)
        syserr("close");
}
```

Zwei knifflige Dinge sind in dieser Funktion zu beachten: Da **read** implizit eine Vorwärtsbewegung in der Datei ausführt, müssen wir jeweils 2 Bytes rückwärts gehen, um das vorhergehende Byte zu lesen. Zweitens gibt es keinen Resultatwert, der den Dateianfang kennzeichnet, so wie es beim Dateiende der Fall ist. Deshalb müssen wir den Positionszeiger (die Variable **where**) im Auge behalten und aufhören, nachdem wir das erste Byte gelesen haben. Alternativ dazu könnten wir warten, bis **lseek** versucht, den Positionszeiger negativ zu machen. Dies ist jedoch unklug, da der Fehler-Code, den man in dieser Situation erhält (**EINVAL**), auch dazu benützt wird, andere ungültige Argumente anzuzeigen.

2.12 Portabilität

Die in diesem Kapitel eingeführten Systemaufrufe sind derart fundamental, daß Portabilität unter UNIX-Versionen kein großes Problem darstellen sollte. Wenn man natürlich zu sehr auf die Besonderheiten einer bestimmten Implementierung vertraut (z.B. Werte von Bytes, die bei einem Schreibzugriff hinter das Dateiende eingefügt wurden), so wird es ganz bestimmt Probleme geben.

Falls Sie Portierungen auf Systeme, die auf Version 7 basieren, voraussehen, so sollten Sie, falls möglich, die neue Form von **open** (mit drei Argumenten) vermeiden. Jedoch wird die neue Form in 4.2 BSD unterstützt.

Die schwerste Entscheidung ist, was man beim Schreiben von portierbarer Software tun soll, wenn verlangt wird, daß Schreibvorgänge physikalisch beendet sein müssen, bevor **write** zurückkehrt. Xenix 3 bietet die vorteilhafteste Methode, die **O_WSYNC**-Flagge, doch nüchtern betrachtet ist es für kommerzielle Software wahrscheinlich unerläßlich, auf AT&T System III und System V portierbar zu sein. AT&T hat diese Xenix-Fähigkeit nicht übernommen, und sie ist auch nicht im vorgeschlagenen Standard von */usr/group* enthalten. Somit verbleibt als einzige portable Methode die Verwendung nackter Dateien, obwohl dies einen wesentlich höheren Entwicklungsaufwand erfordert. In Abschnitt 3.3 werden wir sehen, wie man nackte Dateien verwendet.

Aufgaben

2.1 Modifizieren Sie **lock** im Abschnitt 2.5 so, daß der Login-Name in der Sperrdatei gespeichert wird (Verwenden Sie die Standardfunktion **getlogin**). Fügen Sie ein Argument hinzu, das verwendet werden kann, um den Login-Namen des Benutzers, der in den Besitz der Sperre gekommen ist, zu beschaffen, falls **lock** den Wert **FALSE** zurückliefert.

2.2 Schreiben Sie ein Programm, das eine Datei mit der **O_APPEND**-Flagge zum Schreiben öffnet und dann eine Zeile Text in die Datei schreibt. Lassen Sie

mehrere Prozesse nebeneinander dieses Programm ausführen, um sich davon zu überzeugen, daß die Textzeilen nicht miteinander vermischt werden. Schreiben Sie dann das Programm nochmals ohne **O_APPEND** und verwenden Sie **lseek**, um vor jedem **write** an das Dateiende zu gelangen. Führen Sie erneut die parallelen Prozesse aus, um zu erkennen, ob die Texte nun durcheinandergeraten.

2.3 Lassen Sie die Zeittests für die gepufferte Ein- und Ausgabe mit den Puffergrößen 2, 57, 128, 256, 511, 513 und 1024 laufen. Wenn Sie möchten, versuchen Sie es mit anderen interessanten Zahlen. Falls Sie Zugriff auf mehrere UNIX-Versionen besitzen, lassen Sie das Experiment auf jeder Version laufen und stellen Sie die Ergebnisse in einer Tabelle zusammen.

2.4 Erweitern Sie das **STREAM**-Paket so, daß man eine Datei gleichzeitig zum Lesen und Schreiben öffnen kann.

2.5 Fügen Sie eine Funktion **Sseek** zum **STREAM**-Paket hinzu.

2.6 Schreiben Sie ein B-Baum-Paket für UNIX. Sie werden Funktionen zum Erzeugen einer B-Baum-Datei, zum Öffnen einer existierenden Datei, zum Schließen einer Datei und zum Speichern, Holen und Löschen von Sätzen benötigen. Da ein wesentlicher Vorteil von B-Bäumen darin besteht, daß die Sätze sortiert gespeichert sind, sollten Sie außerdem Funktionen haben, die den ganzen Baum absuchen. Zusätzliche Informationen über B-Bäume finden Sie in "The Ubiquitous B-Tree" von Douglas Comer, erschienen in *Computing Surveys* 11, no. 2 (June 1979), 121–137 und in *The Art of Computer Programming, Vol.3: Sorting and Searching* von D.E. Knuth, erschienen bei Addison-Wesley Publishing Company, Inc., Reading, Mass., 1973.

2.7 Wie Aufgabe 2.6, doch entwickeln Sie Ihren Platten-Änderungs-Algorithmus so, daß er vorsichtige Austauschtechniken benützt (siehe Abschnitt 2.7).

Kapitel 3: Fortgeschrittene Ein- und Ausgabe

3.1 Einführung

Dieses Kapitel macht dort weiter, wo Kapitel 2 aufgehört hat. Zuerst dehnen wir die Anwendung der bereits eingeführten Systemaufrufe auf Kataloge und Platten-Gerätedateien aus. Danach führen wir weitere Systemaufrufe ein, die es uns erlauben, neue Kataloge (und andere Dinge) zu erzeugen, auf bestehende Dateien zu verweisen und Statusinformation von Dateien zu erhalten oder zu ändern.

Sie werden die in diesem Kapitel zusammengestellten fortgeschrittenen Möglichkeiten wohl wesentlich seltener benützen als die aus Kapitel 2. Dennoch werden Sie allein aus dem Wissen profitieren, wie man sie anwendet, da Sie dadurch die Arbeitsweise der Ein- und Ausgabe für Dateien vollständiger verstehen.

Die Programmbeispiele in diesem Kapitel sind umfangreicher als die bisher gezeigten. Es lohnt sich, diese Beispiele sorgfältig zu studieren, da sie Einzelheiten zeigen, die im Text nicht explizit erklärt sind.

3.2 Ein- und Ausgabe für Kataloge

Erinnern Sie sich daran, daß Kataloge mit gewöhnlichen Dateien identisch sind bis auf die Tatsache, daß sie in ihrer Inode ein spezielles Bit gesetzt haben, und daß der Kern Schreibzugriffe auf Kataloge verbietet. Jeder beliebige Systemaufruf zur Dateimanipulation darf ungehindert auf einen Katalog angewendet werden, vorausgesetzt, daß der Inhalt des Katalogs nicht geändert wird. Verboten sind **creat** und **open** mit den Flaggen **O_WRONLY**, **O_RDWR** oder **O_TRUNC**, sowie **write**. Das bedeutet jedoch nicht, daß andere Systemaufrufe sinnvoll sind, sondern nur, daß sie zulässig sind. Zum Beispiel ist es wenig sinnvoll, **lseek** auf einen Katalog anzuwenden, da die Verweise nicht in einer speziellen Reihenfolge angeordnet sind.

Typischerweise öffnet man einen Katalog zum Lesen und liest dann nacheinander die Verweise. Genau das macht das **ls**-Kommando. Als Hilfe bei der Interpretation eines Kataloginhalts gibt es die Definitionsdatei **/usr/include/sys/dir.h**, die die Struktur eines Verweises (**struct direct**) definiert: ein zwei Byte langes Wort, das eine Inumber enthält, gefolgt von einem 14 Byte langen Dateinamen.[1] Der Dateiname ist mit Nullbytes aufgefüllt, falls er kürzer ist als 14 Bytes, so daß er in diesem Falle wie eine mit einem Nullbyte abgeschlossene Zeichenkette aus C behandelt werden kann. Unglücklicherweise ist der Dateiname jedoch nicht mit einem Nullbyte abgeschlossen, wenn er genau 14 Bytes lang ist. Deshalb ist die in **dir.h** zur Verfügung gestellte Struktur unangenehmer zu benützen als eine, die Platz für ein Nullbyte hinter dem Dateinamen bietet. Dies wird durch die folgende Funktion veranschaulicht, die einen Katalog liest und ausgibt:

```
#include <sys/dir.h>

directory(path)  /* Katalog zeigen */
    char * path;
{
    struct directx
    {
        ino_t d_ino;
        char d_name[DIRSIZ + 1];
    } dlink;
    int fd, nread;
    char * dname;

    if ((fd = open(path, 0)) == -1)
        syserr(path);
    dlink.d_name[DIRSIZ] = '\0';
    while ((nread = read(fd, &dlink, sizeof(struct direct)))
          == sizeof(struct direct))
    {
        if (dlink.d_ino == 0)
            dname = "--- UNUSED ---";
        else
            dname = dlink.d_name;
        printf("%-14s %4d\n", dname, dlink.d_ino);
    }
```

[1] Bei UNIX-Systemen, die von Berkeley-Versionen abstammen, haben Kataloge eine vollkommen andere Struktur. A.d.Ü.

```
    switch (nread)
    {
        case 0:
            return;
        case -1:
            syserr("read");
        default:
            fatal("partial read");
    }
}
```

Die Struktur **directx** entspricht genau der offiziellen (**direct**), mit der Ausnahme, daß der Vektor **d_name** um ein Byte länger ist. Wir passen auf, daß wir pro Verweis **sizeof(struct direct)** Bytes lesen, anstatt **sizeof(struct directx)**, da sonst zuviel gelesen werden würde. Beachten Sie, daß eine Inumber mit dem Wert Null einen unbenützten Verweis anzeigt.

Nachfolgend die Ausgabe eines Testlaufs:

```
.                   678
..                  578
memos               676
data                681
--- UNUSED ---       0
AugustAccounts      682
```

Beachten Sie die Einträge für . und .., die automatisch eingerichtet werden, wenn der Kern einen neuen Katalog erzeugt. Da . einen Verweis von diesem Katalog auf sich selbst darstellt, muß dieser Katalog die Inode 678 sein. Sein Vater ist die Inode 578. Den Namen des Vater-Katalogs kennen wir nicht; diese Information findet man nur beim Vater des Vaters.

Der **unlink**-Systemaufruf arbeitet mit Katalogen genauso wie mit gewöhnlichen Dateien, ausgenommen daß er nur von einem Prozeß mit der effektiven Benutzernummer 0 (dem Super-User) abgesetzt werden darf. Damit soll das Löschen eines nichtleeren Katalogs vermieden werden, wodurch Waisendateien erzeugt würden, d.h. Dateien, die zwar existieren, auf die aber in keinem Katalog verwiesen wird.[2] Vom Super-User wird angenommen, daß er weiß, was er tut.

Von gewöhnlichen Benutzern wird erwartet, daß sie sich des **rmdir**-Kommandos bedienen, anstatt **unlink** direkt zu benützen. Dieses Kommando ist so programmiert, daß wirklich nur leere Kataloge entfernt werden. Die Datei, die das **rmdir**-Kommando beherbergt, gehört dem Super-User, und das *set-user-ID*-Bit ist ge-

[2] Das **fsck**-Programm, das Dateisysteme repariert, fungiert als Adoptionsbüro. Es bindet verwaiste Dateien in den Katalog **lost+found** ein, wobei es die Inumber als Dateinamen verwendet.

setzt. Wenn daher ein gewöhnlicher Benutzer dieses Kommando ausführt, wird die effektive Benutzernummer auf die des Super-Users gesetzt.

Um einen Katalog zu erzeugen, wird der Systemaufruf **mknod** verwendet. Er entspricht **creat** bei gewöhnlichen Dateien. **link** und **unlink** ändern den Inhalt eines Katalogs: **link** erzeugt einen Verweis, **unlink** entfernt einen. **link** und **mknod** werden in den Abschnitten 3.6 und 3.8 besprochen.

Kataloge sind im wesentlichen gewöhnliche Dateien. Dateien kann man auf keine andere Größe als auf Null verkürzen. Folglich ist es nicht möglich, den Plattenplatz direkt wiederzugewinnen, der ursprünglich von einem umfangreichen Katalog belegt war, der nun viele unbenutzte Verweise enthält. UNIX-Benutzer wissen das nicht immer. Manchmal glaubt ein Benutzer (zu Recht), daß ein sehr umfangreicher Katalog den Zugriff auf Dateien verlangsamt und meint (zu Unrecht), daß es hilft, die meisten Dateien in untergeordnete Kataloge zu stecken. Um einen Katalog tatsächlich wirksam zu verkleinern, muß man einen Geschwister-Katalog erzeugen, alle Verweise in den neuen Katalog verschieben, den alten Katalog entfernen und schließlich den neuen Katalog auf den Namen des alten Katalogs umtaufen. Es gibt keinen Systemaufruf, um Verweise zu "verschieben", aber es gibt natürlich entsprechende Kommandos (**mv** und **mvdir**). Wie wir in Abschnitt 3.6 sehen werden, wird auf der Ebene der Systemaufrufe das Verschieben durch **link** und **unlink** vorgenommen.

3.3 Ein- und Ausgabe für Platten-Gerätedateien

Bis jetzt haben wir Ein- und Ausgabe ausschließlich durch das Dateisystem des Kerns durchgeführt, wobei wir mit ziemlich abstrakten Gebilden wie Datei, Katalog und Inode gearbeitet haben. Wie in Abschnitt 2.7 erklärt, baut das Dateisystem auf dem Block-System der Ein- und Ausgabe auf, das das Blockdepot benützt. Das Block-System der Ein- und Ausgabe wird durch eine Block-Gerätedatei, kurz Blockgerät, angesprochen, das eine direkte Verbindung zur Platte darstellt. Die Platte wird als Folge von Blöcken fester Größe (typischerweise 512 Bytes) betrachtet, die, mit 0 beginnend, der Reihe nach durchnumeriert sind.

Es mag mehrere physikalische Platten geben und jede dieser physikalischen Platten kann in Stücke aufgeteilt sein, die *logische Platten* oder auch *Dateisystem* genannt werden. Der Begriff *Dateisystem* ist verwirrend, da er auch einen Teil des Kerns beschreibt. Der Zusammenhang, in dem wir den Begriff verwenden werden, wird die beabsichtigte Bedeutung klarstellen.

Jede logische Platte entspricht einer Gerätedatei, deren Name aus einem Gerätenamen wie **rp** oder **hd**, gefolgt von einer Zahl oder einem Buchstaben, gebildet wird. Diese Gerätedateien sind stets im Katalog **/dev** eingebunden, obwohl dies nicht so sein muß. Die Platten-Gerätedateien können zum Beispiel mit **/dev/hd0**, **/dev/hd1** und so weiter bis zu **/dev/hd7** benannt sein.

Im Prinzip kann auf eine Platten-Gerätedatei mit Ein- und Ausgabe-Systemaufrufen genauso zugegriffen werden, als ob sie eine gewöhnliche Datei wäre. Sie kann zum Lesen und/oder zum Schreiben geöffnet werden, (in willkürlichen Einheiten) gelesen oder beschrieben werden; man kann sich in ihr (zu jeder beliebigen Byte-Position) bewegen, und sie kann geschlossen werden. Das Blockdepot wird benützt (da es sich um eine Block-Gerätedatei handelt), doch innerhalb der logischen Platte gibt es keine Kataloge, Dateien, Inodes, Zugriffsrechte, Besitzer, Längenangaben, Zeitstempel und so weiter. Man befaßt sich einfach mit einem riesigen Vektor durchnumerierter Blöcke.

In der Praxis ist es den meisten Personen nicht möglich, eine dieser Operationen durchzuführen, da ihnen dafür die Rechte fehlen. Das Lesen einer Platte, die Dateien anderer Benutzer enthält, verletzt deren Privatsphäre, selbst wenn die Platte als planloses Durcheinander erscheint, wenn man sie als Gerätedatei betrachtet (Blöcke sind UNIX-Dateien und Katalogen in keiner bestimmten Reihenfolge zugeordnet). Und das Beschreiben einer Platte, ohne das Dateisystem des Kerns zu benützen, würde sogar noch größere Verwüstungen hinterlassen.

Wenn andererseits eine logische Platte für einen Benutzer reserviert ist und nicht für Dateien und Kataloge anderer Benutzer verwendet wird, dann gibt es keine Konflikte. In der Tat möchten Benutzer, die Datenbank-Managementsysteme oder Datenerfassungs-Systeme implementieren, eine logische Platte reserviert haben, so daß sie sie als Block-Gerätedatei ansprechen können. Dies erlaubt schnellere Ein- und Ausgabe. Eine Beschränkung ist natürlich dadurch gegeben, daß es nur endlich viele Platten-Gerätedateien gibt. Üblicherweise ist eine Anwendung, die eine Platten-Gerätedatei benützt, die einzige oder zumindest die wichtigste Anwendung auf dem Computer. Noch viel schneller als Platten-Gerätedateien sind *nackte* Platten-Gerätedateien. Diese Gerätetreiber befassen sich mit denselben Plattenbereichen, und sogar ihre Namen sind ähnlich: üblicherweise wird ein **r** vorangestellt (z.B. **/dev/rhd0**, **/dev/rhd1** usw). Diese nackten Gerätedateien sind jedoch *Character*-Gerätedateien (zeichenorientiert), keine Block-Gerätedateien. Das heißt, daß sie nicht dem Blockmodell unterliegen, und daß sie nicht das Blockdepot benützen. Sie machen sogar etwas Besseres.

Wird ein **read**- oder ein **write**-Aufruf auf eine nackte Gerätedatei eingeleitet, so wird der Prozeß im Speicher gesperrt (ein Auslagern (*Swappen*) wird verhindert), so daß sich keine physikalische Datenadresse ändern kann. Dann wird die Platte beauftragt, Daten via *DMA* (*direct memory access*) zu übertragen. Die Daten fließen nun direkt zwischen dem Datensegment des Prozesses und dem Platten-Controller, ohne daß sie durch den Kern gehen. Die Übertragungsmenge darf auch mehr als einen Block auf einmal betragen.

Im allgemeinen ist Ein- und Ausgabe für nackte Geräte weniger anpassungsfähig als für gewöhnliche Dateien oder Block-Gerätedateien. Es kann sein, daß Ein- und Ausgabe nur in Vielfachen der Blockgröße funktioniert. Es ist auch möglich, daß die DMA-Hardware fordert, daß die Pufferadresse des Prozesses auf einer Wort-

kante liegt. Weiterhin kann es sein, daß Positionierbewegungen nur auf Blockkanten möglich sind. Diese Beschränkungen bedeuten Entwicklern von datenbankorientierten Dateisystemen wahrscheinlich wenig, da sie es normal finden, daß eine Platte logisch irgendwie aus Seiten fester Größe besteht.

Bisher haben wir beobachtet, daß die Fähigkeiten von UNIX von Version zu Version etwas verschieden sind. Nun haben wir jedoch eine Abweichung, die auch vom *Kanal* abhängt. Da Computer sich in ihrer Ein- und Ausgabe-Hardware stark unterscheiden, unterscheiden sich entsprechend auch die Gerätetreiber. Außerdem beeinflussen die jeweiligen Aufgaben, die an den Kanal gestellt werden, natürlich auch die Wichtigkeit einer schnellen nackten Ein- und Ausgabe. Auf einem Mehrzweck-Tischcomputersystem zum Beispiel mag dies von sehr geringer Bedeutung sein.

Nackte Ein- und Ausgabe besitzt einen beachtlichen Vorteil, aber auch einen gewaltigen Nachteil. Der Vorteil: Da ein Prozeß wartet, bis ein **write**-Aufruf beendet ist und da kein Zweifel darüber besteht, welchem Prozeß die Daten gehören, kann ein Prozeß durch einen Resultatwert von –1 und einen **errno**-Fehler-Code über physikalische Schreibfehler unterrichtet werden. Dafür gibt es einen Fehler-Code (**EIO**), doch ob er jemals übergeben wird, bleibt dem Entwickler des Gerätetreibers überlassen. Die meisten Entwickler unterstützen diesen Code, doch da die meisten UNIX-Handbücher kaum mehr als eine Reproduktion des von AT&T herausgegebenen Handbuchs sind, enthalten sie selten genügend Einzelheiten über nackte Ein- und Ausgabe für spezielle Hardware-Konfigurationen. Am besten schaut man sich den Quellcode des Gerätetreibers an, sofern dieser verfügbar ist (meistens ist er es nicht), oder man fragt den Händler. Ausprobieren ist eine weniger gute Methode, um den Fehlermeldungsmechanismus zu erforschen, da es zu schwierig ist, Ein- und Ausgabe-Fehler zu erzeugen, ganz abgesehen von der Reproduzierbarkeit.

Nun zum Nachteil: Da der Prozeß wartet, bis ein **read**- oder **write**-Aufruf ganz beendet ist und da die Konstruktion von UNIX einem Prozeß nur gestattet, jedesmal einen einzelnen Systemaufruf abzusetzen, erfolgt die Ein- und Ausgabe zwar schnell, doch nicht sehr oft. Zum Beispiel gibt es bei einer Mehrbenutzer-Datenbankanwendung mit einem zentralen Datenbank-Managerprozeß (eine weitverbreitete Anordnung) eine gigantische Verkehrsstockung bei der Ein- und Ausgabe, wenn diese nackt verwendet wird, da dann nur ein einziger Prozeß die Arbeit für alle anderen macht. Doch auch Ein- und Ausgabe für gewöhnliche Dateien ist nicht viel wünschenswerter, so daß die Situation auf den ersten Blick hoffnungslos erscheint. Hierin liegt ein Grund für den Ruf von UNIX, für Datenbankanwendungen ungeeignet zu sein.

Der einzige Weg, das Verkehrschaos zu mildern (abgesehen von Änderungen am Kern) besteht darin, daß der zentrale Datenbank-Managerprozeß mehrere *Drohnen*-Prozesse erzeugt, die die tatsächliche Ein- und Ausgabe vornehmen. Anstatt daß der zentrale Prozeß selbst liest oder schreibt, reicht er die Anfrage über Prozeßkommunikation an einen Drohnen-Prozeß weiter, der dann den Lese- oder Schreib-Aufruf absetzt und wartet. Der zentrale Prozeß kann später den Drohnen-

Prozeß befragen, um herauszufinden, ob die Ein- und Ausgabe beendet ist. Mit anderen Worten: Nachdem wir die Tätigkeit des Datenbankmanagers zentralisiert haben, dezentralisieren wir die tatsächliche Ein- und Ausgabe. Das Problem an dieser Methode ist, daß dadurch der Aufwand an Prozeßkommunikation enorm wächst. Bei älteren UNIX-Versionen wurde dadurch ein weiterer Flaschenhals geschaffen, doch seit System V oder Xenix 3 ist dieser Weg gangbar (siehe Kapitel 7). Die Programmierung solcher Hochleistungs-Datenbank-Managementsysteme ist jedoch alles andere als einfach.

Ein weniger bedeutender Nachteil nackter Ein- und Ausgabe, den wir sofort aus dem Weg räumen können, ist folgender: Solange ein Prozeß im Speicher gesperrt ist, kann es eventuell nicht möglich sein, andere Prozesse auszuführen, da nicht genug Platz ist, sie in den Speicher zu bringen. Dies wäre ein Jammer, da die CPU durch DMA nicht benutzt wird, und somit diese Prozesse sonst tatsächlich ablaufen könnten. Die Lösung besteht darin, einfach mehr Speicher einzubauen. Bei den heutigen Preisen (ungefähr ein Zehntel Penny pro Byte)[3] gibt es keine Entschuldigung dafür, daß man nicht genügend Speicher hat, um das Auslagern von Prozessen größtenteils oder gar ganz zu vermeiden — besonders auf einem Computer, auf dem eine Anwendung läuft, die wichtig genug ist, daß nackte Ein- und Ausgabe durchgeführt wird.

Die nachfolgende Tabelle zählt die funktionellen Unterschiede auf, die zwischen Ein- und Ausgabe für gewöhnliche Dateien, für Platten-Gerätedateien und für nackte Platten-Gerätedateien bestehen:

Merkmal	*Gewöhnliche Datei*	*Platten-Gerätedatei*	*nackte Platten-Gerätedatei*
Kataloge, Dateien, Inodes			
Zugriffsschutz	Ja	Nein	Nein
Blockdepot	Ja	Ja	Nein
E-/A-Fehler-Resultate, DMA	Nein	Nein	Ja

Um die Geschwindigkeitsunterschiede zu zeigen, stoppten wir die Zeit für eine Schleife, in der 204800 Bytes in 100 Portionen zu je 2048 Bytes gelesen wurden. Wir lasen eine gewöhnliche Datei, eine Platten-Gerätedatei (**/dev/hd0**) und eine nackte Platten-Gerätedatei (**/dev/rhd0**). Wie bereits in Abschnitt 2.10 zeigen wir die *Real-Time*, die *User-Time* und die *System-Time*, angegeben in Sekunden:

[3] Heute weniger als ein Zehntel Pfennig. A.d.Ü.

Dateiart	*Real-Time*	*User-Time*	*System-Time*
Gewöhnliche Datei	10.77	0.00	4.60
Platten-Gerätedatei	6.40	0.02	2.63
nackte Platten-Gerätedatei	4.13	0.02	0.58

Wie Sie sehen können, ist der Geschwindigkeitsvorteil bei nackter Ein- und Ausgabe enorm. Wir hielten uns nicht mit einem Vergleich für Schreibzugriffe auf, da wir wissen, daß die zwei Dateitypen, die das Blockdepot benützen, spielend gewonnen hätten, und da wir auch keine freie logische Platte hatten, auf der wir herumkritzeln konnten. Natürlich wäre es kein fairer Vergleich gewesen, da die Zeiten bei Benutzung des Blockdepots nicht die physische Ein- und Ausgabe beinhaltet hätten, und da die nackte Ein- und Ausgabezeit nicht viel ausgesagt hätte.

Es ist aufschlußreich, die interne Struktur einer logischen Platte zu betrachten, indem man sie als Platten-Gerätedatei liest (vorausgesetzt, man hat Leserecht). Block 0 (der erste Block) eines Dateisystems ist unbenützt. Block 1 wird *Super-Block* genannt; er enthält verschiedenartige strukturelle Informationen wie die Zahl der Inodes, die Zahl der Blöcke auf der logischen Platte, den Anfang der verketteten Liste der freien Blöcke und so weiter. Eine C-Struktur, die den Super-Block beschreibt, steht in **/usr/include/sys/filsys.h** zur Verfügung.

Die Iliste, die aus einer Folge von 64-Byte-Inodes besteht, beginnt mit Block 2. Inodes werden mit 1 beginnend durchnumeriert, doch die Inode 1 selbst wird für keine wirkliche Datei verwendet. Stattdessen kann man sie für eine Pseudo-Datei verwenden, die kaputte Plattenblöcke enthält, um zu verhindern, daß diese an die freie Liste oder an eine Datei zugewiesen werden. Inode 2 ist für den Wurzelkatalog (/) reserviert, so daß der Kern bei einer ihm bekannten Stelle beginnen kann, wenn er einen absoluten Pfadnamen erhält. Keine andere Inumber hat eine spezielle Bedeutung; alle werden nach Bedarf an Dateien zugewiesen. Alle Blöcke, die hinter der Iliste übrigbleiben, stehen für Datenblöcke zur Verfügung. Am Anfang sind sie in der freien Liste eingehängt. Wenn eine Datei anwächst, werden Blöcke aus der freien Liste herausgenommen und an die Datei zugewiesen. Wenn eine Datei (auf Länge 0) verkürzt oder weggeworfen wird, so werden ihre Blöcke wieder in die freie Liste eingehängt.

Die Definitionsdatei **/usr/include/sys/ino.h** enthält die C-Struktur für eine Inode (**struct dinode**). Die folgende Funktion gibt Informationen über die ersten fünf Inodes aus:

```
#include <sys/types.h>
#include <sys/ino.h>

inodes()                              /* Inode-Information ausgeben */
{
    struct dinode di;
    int fd, inumber, nread;
    long lseek();

    if ((fd = open("/dev/hd0", 0)) == -1)
        syserr("open");
                                      /* auf Inode 2 positionieren  */
    if (lseek(fd, 1024L + sizeof(di), 0) == -1)
        syserr("lseek");
    for (inumber = 2; inumber <= 6; inumber++)
    {
        if ((nread = read(fd, &di, sizeof(di))) != sizeof(di))
            switch (nread)
            {
                case 0:
                    fatal("EOF");
                case -1:
                    syserr("read");
                default:
                    fatal("partial read");
            }
        printf("i-node %d; owner = %3o; size = %7ld\n",
                inumber, di.di_uid, di.di_size);
    }
}
```

Nun die Ausgabe, die das Programm erzeugt hat:

```
i-node 2; owner=   2; size=     256
i-node 3; owner=   0; size=      27
i-node 4; owner=   2; size=    1280
i-node 5; owner=   2; size=     744
i-node 6; owner=   2; size=   11182
```

Um Informationen aus einer Inode herauszubekommen, müssen Sie die Platten-Gerätedatei nicht direkt ansprechen, wie wir es taten. Man kann auch den Systemaufruf **stat** verwenden, den wir in Abschnitt 3.12 erklären werden.[4]

[4] Dazu muß man den Namen der Datei kennen. A.d.Ü.

3.4 Datum und Uhrzeit

Unter UNIX werden gewöhnlich zwei Zeitbegriffe verwandt: *Ausführungszeit* und *Kalenderzeit*. Ausführungszeit wird zum Messen von Zeitabständen (wie wir es im vorigen Abschnitt gemacht haben) und zur Buchführung verwendet. Der Kern zeichnet automatisch die Ausführungszeit jedes Prozesses auf; mit dem Systemaufruf **times** (siehe Abschnitt 9.10) kann man sie erhalten. Ausführungszeiten werden in Einheiten von *ticks* gemessen, die von Computer zu Computer verschieden sind (60 oder 100 *ticks* pro Sekunde).

Die Kalenderzeit wird in Form der seit Mitternacht am 1. Januar 1970, GMT (*Greenwich Mean Time* — westeuropäische Zeit) verstrichenen Sekunden geführt. Sie wird für die Zugriffs-, Änderungs- und Statusänderungszeitstempel von Dateien verwendet, zur Aufzeichnung, wann sich ein Benutzer angemeldet hat und so weiter. Die GMT stellt außerdem eine gute Möglichkeit für Anwendungsprogramme dar, Zeiten in Datendateien zu speichern, doch manchmal wird stattdessen die Ortszeit verwandt. Da der Systemaufruf **time** (siehe Abschnitt 9.11) eine long-Integer liefert, muß dieser Wert in 31 Bits passen; diese reichen jedoch aus, die Sekundenzahl bis zum 19. Januar 2038 aufzunehmen, und bis dahin hat sicherlich jemand die Stunde 0 verlagert.

Der UNIX-Kern gibt sich mit keinen anderen Zeitzonen außer der GMT ab, bei den Benutzern ist das anders. Bei der Ein- oder Ausgabe von Zeiten ist es eben üblich, die Ortszeit zu verwenden, was die Situation erschwert. Es muß für den Computer einen Weg geben, herauszubekommen, in welcher Zeitzone sich der Benutzer befindet und ob gerade Normalzeit oder Sommerzeit herrscht. Und was noch schlimmer ist, wenn der Computer frühere Zeitangaben ausgibt, so muß er wissen, ob damals Normal- oder Sommerzeit war. Es gibt eine Sammlung von Unterprogrammen, genannt **ctime**, die versucht, dieses Problem zu bewältigen.[5] Zeitzonen werden korrekt behandelt; Normalzeit und Sommerzeit werden zwar unvollkommen, jedoch gut genug verarbeitet.

Die Zeitzone eines Benutzers ist in der Environment-Variablen **TZ** abgelegt, welche durch den Systemadministrator richtig gesetzt sein muß (nur ein einziges Mal, vorausgesetzt der Computer muß nicht umziehen). Die ersten drei Buchstaben des Wertes von **TZ** geben den Namen der Zeitzone an, die letzten drei den Namen während der Sommerzeit (falls sie wahrgenommen wird). Die Buchstaben in der Mitte sind der Ausgleich zur GMT in Stunden, positiv für Gebiete westlich von Greenwich und negativ für Gebiete östlich davon. **MST7MDT** ist ein Beispiel für Colorado.

Eine kniffligere Frage stellt sich darin, ob eine bestimmte Zeitangabe Sommerzeit oder Normalzeit darstellt. Als dies geschrieben wurde,[6] begann die Sommerzeit

[5] Das Paket **ctime**, das die Funktionen **localtime** und **asctime** beinhaltet, ist im Abschnitt 3 des UNIX-Handbuchs beschrieben.

[6] aber nicht diese Übersetzung... A.d.Ü.

um 2 Uhr morgens am letzten Sonntag im April, und endete am letzten Sonntag im Oktober. Doch dies trifft nur auf die Vereinigten Staaten zu, und dort nicht für Arizona und Hawaii. Um die Dinge noch weiter zu komplizieren, wurde die Sommerzeit 1974 und 1975 auch im Winter und zu Beginn des Frühjahrs eingehalten (um Energie einzusparen). Eine Tabelle im **ctime**-Paket beinhaltet diese historische Information. Wenn eine Zeitzone die Sommerzeit einhält, was durch die Variable **TZ** angezeigt wird, so nimmt **ctime** an, daß dies schon immer so war, und es wird die interne Tabelle benützt. Dies läßt die Staaten und Länder unberücksichtigt, die ihre Politik der Einhaltung der Sommerzeit ändern. Glücklicherweise sind wir gewöhnlich schon damit zufrieden, wenn wir eine frühere Zeitangabe bis auf eine Stunde genau kennen. *Vergleiche* zwischen Zeiten sind kritischer, doch bereitet dies hier keine Probleme, da die Vergleiche innerhalb des Computers stattfinden, wo nur die GMT benutzt wird.

Die **ctime**-Unterprogramme helfen nur bei der Ausgabe von Zeiten, jedoch nicht bei der Eingabe, und es gibt auch keine standardisierten UNIX-Unterprogramme, die dies tun. Das Übersetzen eines Datums aus einer üblichen Darstellung wie Jahr-Monat-Tag-Stunde-Minute-Sekunde (YYMMDDhhmmss) in die Zahl der seit 1970 verstrichenen Sekunden ist überraschend kompliziert. Man muß die unterschiedliche Anzahl von Tagen pro Monat berücksichtigen, außerdem Schaltjahre, Zeitzonen und die Einhaltung der Sommerzeit. Um zu zeigen, wie man diese Umwandlung vornimmt, präsentieren wir nun ein Programm, das eine 12-stellige Zahl der Form YYMMDDhhmmss in die interne Darstellung umwandelt:

```
#include <time.h>

static long timecvt(atime)  /* YYMMDDhhmmss in interne  */
    char * atime;           /* Darstellung konvertieren */
{
    long tm;
    int i, n, days, tzone;
    char s[3], * getenv(), * tz;
    BOOLEAN isleapyear;
    extern struct tm * localtime();

    if (strlen(atime) != 12)
    {
        printf("Time must be 12 digits (YYMMDDhhmmss)\n");
        return(0);
    }
```

```
for (i = 0; i < 12; i++)
    if (atime[i] < '0' || atime[i] > '9')
    {
        printf("Time contains non-numeric character\n");
        return(0);
    }
if ((tz = getenv("TZ")) == NULL)
    tzone = 5;                      /* Eastern USA Zeitzone          */
else
    tzone = atoi(&tz[3]);
s[2] = '\0';
for (i = 0; i < 12; i += 2)
{
    s[0] = atime[i];
    s[1] = atime[i + 1];
    n = atoi(s);
    switch (i)
    {
        case 0:                     /* YY: Jahr                      */
                                    /* 2000 ist keins                */
            isleapyear = n != 0 && n % 4 == 0;
            if (n < 70)
                n += 100;           /* Jahre nach 2000               */
            days = (n - 70L) * 365L;
                                    /* Schalttage vergangener Jahre */
            days += ((n - 69L) / 4);
            continue;
        case 2:                     /* MM: Monat                     */
            switch (n)
            {
                case 12:
                    days += 30;  /* November                         */
                case 11:
                    days += 31;  /* Oktober                          */
                case 10:
                    days += 30;  /* September                        */
                case 9:
                    days += 31;  /* August                           */
                case 8:
                    days += 31;  /* Juli                             */
```

```
            case 7:
                days += 30;  /* Juni                          */
            case 6:
                days += 31;  /* Mai                           */
            case 5:
                days += 30;  /* April                         */
            case 4:
                days += 31;  /* Maerz                         */
            case 3:
                             /* Februar                       */
                days += (isleapyear ? 29 : 28);
            case 2:
                days += 31;  /* Januar                        */
            case 1:
                break;
            default:
                printf("Invalid month\n");
                return(0);
        }
        continue;
    case 4:                  /* DD: Tag                       */
        if (n > 31)
        {
            printf("Invalid day\n");
            return(0);
        }
        tm = (days + n - 1) * 24L * 60L * 60L;
        continue;
    case 6:                  /* hh: Stunde                    */
        if (n > 23)
        {
            printf("Invalid hour\n");
            return(0);
        }
        n += tzone;          /* um Zeitzone korrigieren       */
        tm += n * 60L * 60L;
        continue;
```

```
            case 8:                 /* mm: Minute                    */
                if (n > 59)
                {
                    printf("Invalid minute\n");
                    return(0);
                }
                tm += n * 60L;
                continue;
            case 10:                /* ss: Sekunde                   */
                if (n > 59)
                {
                    printf("Invalid second\n");
                    return(0);
                }
                tm += n;
        }
        if (localtime(&tm)->tm_isdst)
            tm -= 60L * 60L;        /* um Sommerzeit korrigieren     */
        return(tm);
    }
}
```

Die Standardfunktion **getenv** beschafft den Wert einer Environment-Variablen (in diesem Fall TZ).

Wir haben bei der Bestimmung, ob die eingegebene Zeit Normal- oder Sommerzeit war, gemogelt. Wir berechneten die interne Zeit, als ob sie in Normalzeit gegeben wäre, und verlangten dann von **localtime** (Bestandteil des **ctime**-Pakets), für uns herauszufinden, ob zu der betreffenden Zeit tatsächlich Sommerzeit herrschte. Dies stimmt nicht ganz für die Zeit um 2 Uhr morgens an den Umstellungstagen. Wir überlassen diese geringfügige Verbesserung den Lesern, die dies stört, als Übungsaufgabe.

3.5 Dateiattribute

In den Kapiteln 1 und 2 haben wir erläutert, daß die 9 am weitesten rechts stehenden Bits der Attribute einer Datei deren Zugriffsschutz beschreiben, und als Lese-, Schreib- und Ausführungsrecht für den Besitzer, die Gruppe und alle anderen aufgefaßt werden können. Hier betrachten wir die sieben Bits links davon.

Die folgende Tabelle zeigt die Bedeutung jedes einzelnen Bits. Die Bits sind von 15 auf der linken Seite bis 0 auf der rechten Seite durchnumeriert.

Bits	*Symbol*	*Wert*	*Bedeutung*
15–12	**S_IFMT**	0170000	Dateityp:
15	**S_IFREG**	0100000	gewöhnliche Datei
14	**S_IFDIR**	040000	Katalog
13	**S_IFCHR**	020000	Character-Gerätedatei
14–13	**S_IFBLK**	060000	Block-Gerätedatei
12	**S_IFIFO**	010000	FIFO-Datei
11	**S_ISUID**	04000	*set-user-ID*
10	**S_ISGID**	02000	*set-group-ID*
9	**S_ISVTX**	01000	*save-swapped-text*
8–0			Bits für Zugriffsschutz

Die Symbole sind in der Definitionsdatei **/usr/include/sys/stat.h** mit den angegebenen Werten definiert. Ist eine 16-Bit-Attributkombination gegeben, so kann man mit Hilfe dieser Symbole bestimmte Bits ausblenden oder abfragen. Zum Beispiel ist folgender Ausdruck bei Katalogen "wahr":

```
(mode & S_IFMT) == S_IFDIR
```

Sie dürfen die Symbole **S_IFDIR** und **S_IFCHR** nicht als Maske verwenden, da Block-Gerätedateien dadurch gekennzeichnet sind, daß dort beide Bits gesetzt sind. Das heißt, wenn wir den obigen Ausdruck aus Versehen als

```
(mode & S_IFDIR) == S_IFDIR  /* falsch */
```

geschrieben hätten, so würde fälschlicherweise herauskommen, daß eine Block-Gerätedatei ein Katalog ist. Als Masken dürfen diejenigen Symbole verwendet werden, die kein Bestandteil des vier Bit breiten Felds sind, der den Dateityp bestimmt, da sie jeweils nur ein einzelnes Bit bestimmen.

Auch für das neun Bit breite Feld, das den Zugriffsschutz bestimmt, sind einige Symbole definiert, doch ist es einfacher und ebenso lesbar, wenn man die vertrauten Oktalzahlen benützt. Zum Beispiel könnte eine Beschreibung der Dateiattribute für eine FIFO-Datei wie folgt aussehen:

```
S_IFIFO | 0664
```

Die 12 am weitesten rechts stehenden Bits der Dateiattribute können in allen Systemaufrufen benützt werden, die diese Attribute verwenden: **creat**, **open**, **chmod**, **stat**, **fstat** und **mknod**. Die vier am weitesten links stehenden Bits (der Dateityp) werden nur bei **stat**, **fstat** und **mknod** verwendet. Das Bit, das gewöhnliche Dateien kennzeichnet (**S_IFREG**), wird auch durch **creat** und **open** gesetzt.

Die Bedeutung des *save-swapped-text*-Bits (**S_ISVTX**) wurde bisher noch nicht erklärt, so daß wir dies jetzt nachholen wollen. Man kann ein Programm kennzeichnen, daß es *Sharable-Text* oder, ein besserer Ausdruck dafür, *Sharable-Instructions* besitzt. (Manchmal wird dafür auch der Ausdruck *pure-procedure* verwendet.) Die Voraussetzungen für eine solche Markierung sind, daß Anweisungen und Daten des Programms in getrennten Segmenten leben (das heißt, die Daten beginnen bei einer Segmentgrenze), und daß das Anweisungssegment nicht modifiziert wird.

Der Nutzen ist wie folgt: Wenn zwei oder mehr Prozesse gleichzeitig dasselbe Programm ausführen, können sie denselben physikalischen Speicher gemeinsam benützen, in dem das Anweisungssegment untergebracht ist. Dies erreicht der Kern ganz einfach dadurch, daß er die jeweiligen Register für das Anweisungssegment alle auf die gleichen Speicheradressen zeigen läßt. Die Ersparnis beim Ein- und Auslagern von großen, häufig benutzten Programmen wie Texteditoren oder Shells ist beträchtlich.

Das **S_ISVTX**-Bit, manchmal auch das *sticky-Bit* genannt, bewirkt also, daß das Anweisungssegment nur ein einziges Mal ausgelagert wird, und dann im *Swap-Device* verbleibt, selbst nach Prozeßende. Wenn dann ein späterer Prozeß dasselbe Programm ausführt, so kann das Textsegment vom *Swap-Device* geladen werden, anstatt aus der Datei, die das Programm enthält. Dies erhöht die Leistung, wenn das Programm häufig in Anspruch genommen wird, und wenn das *Swap-Device* merklich schneller ist als das Gerät, auf dem das Programm gespeichert ist.[7]

An dieser Stelle (wie auch an jeder anderen) bietet sich die Gelegenheit, einen weiteren Begriff zu erklären, der mit Programmen verbunden ist, die *Sharable-Instructions* benützen: *text-busy*. Solange ein derartiges Programm ausgeführt wird, dürfen seine Anweisungen nicht verändert werden. Falls sie geändert worden wären, so würde der Prozeß, wenn er das nächste Mal eingelagert würde, die *alten* Anweisungen (aus der gemeinsam benutzbaren Kopie) zusammen mit den *neuen* Daten (aus der Programmdatei) benützen, was zu unbeschreiblichen Ergebnissen führen würde. Um so etwas zu vermeiden, weigert sich der Kern, ein Programm aus einer Datei zu laden, die gegenwärtig zum Schreiben geöffnet ist, oder eine Datei zum Schreiben zu öffnen, die ein Programm mit *Sharable-Instructions* enthält, das gerade ausgeführt wird. In beiden Fällen kehrt der zuwiderhandelnde Systemaufruf mit -1 zurück und die Variable **errno** besitzt den Wert **ETXTBSY**.

[7] Im Gegensatz zu Block-Dateien ist *Swap-Space* zusammenhängend (*contiguous*). Es genügt *ein* E/A-Aufruf. A.d.Ü.

3.6 Der Systemaufruf "link"

```
int link(oldpath, newpath)  /* auf Datei verweisen  */
    char * oldpath;         /* existente Datei      */
    char * newpath;         /* neuer Katalogeintrag */

/* liefert 0 bei Erfolg oder -1 bei Fehler          */
```

Der Systemaufruf **link** fügt einen neuen Verweis in einen Katalog ein. Über das erste Argument, *oldpath*, das ein bestehender Verweis sein muß, wird die zu benutzende Inumber ermittelt. Das zweite Argument, *newpath*, gibt den Namen des neuen Verweises an. Beide Verweise sind in jeder Hinsicht gleichwertig, da es in UNIX keine Idee von "ersten" und "zweiten" Verweisen gibt. Die effektive Benutzernummer oder die effektive Gruppennummer muß für den Katalog, der den neuen Verweis enthalten soll, Schreibrecht besitzen. Der durch das zweite Argument gegebene Verweis darf noch nicht existieren; **link** kann man nicht dazu verwenden, bestehende Verweise zu ändern. Existiert **newpath** schon, muß dieser Verweis zuerst mittels **unlink** entfernt werden.

Sieht man von den Verweisen . und .. in jedem Katalog ab, so bildet die Katalogstruktur einen vollkommenen Baum, das heißt, es gibt keine Schleifen. Programme, die diesen Baum rekursiv durchlaufen (wie das **find**-Kommando), können deshalb programmiert werden, ohne daß man befürchten muß, daß ein Katalog zweimal durchlaufen wird, oder daß man sich, was noch schlimmer wäre, in einer Endlosschleife festfährt. Um sicherzustellen, daß diese Regel eingehalten wird, darf nur der Super-User einen zweiten Verweis (außer . und ..) auf einen Katalog etablieren.

Warum muß gerade der Super-User auf einen Katalog zweimal verweisen können? Nun, um zu ermöglichen, daß ein Katalog an eine andere Stelle im Baum verlagert oder ein Katalog umbenennt werden kann. Solange der zusätzliche Verweis schnell wieder entfernt wird, entsteht kaum Schaden.

Daher besteht die allgemeine Strategie für das **mv**-Kommando darin, den neuen Namen zu entfernen, falls dieser bereits existiert, dann einen Verweis vom alten auf den neuen Namen zu machen und schließlich den alten Namen zu entfernen. Wenn wir Super-User spielen, so funktioniert dies auch bei Katalogen, abgesehen davon, daß wir nicht riskieren wollen, den neuen Namen zu entfernen, da dieser Katalog nicht unbedingt leer sein muß. Dies ist in der folgenden Funktion berücksichtigt:

```
mv(from, to)  /* Datei oder Katalog verlagern; nur fuer Super-User */
    char * from, * to;
{
    BOOLEAN isdir();
    extern int errno;
```

```
    if (!isdir(to))
        if(unlink(to) == -1 && errno != ENOENT)
            syserr("unlink1");
    if (link(from, to) == -1)
        syserr("link");
    if (unlink(from) == -1)
        syserr("unlink2");
}

BOOLEAN isdir(path)  /* ist der Pfad ein Katalog? */
    char * path;
{
    BOOLEAN ans;
    int fd;
    extern int errno;

    ans = (fd = open(path, 1)) == -1 && errno == EISDIR;
    if (fd != -1 && close(fd) == -1)
        syserr("close");
    return(ans);
}
```

Der von uns benutzte Trick zum Bestimmen, ob eine Datei ein Katalog ist, besteht in dem Versuch, die Datei zum Schreiben zu öffnen und nachzuprüfen, ob dies mißlang, weil die Datei ein Katalog ist. Eine sauberere Methode wäre, **isdir** neu zu schreiben und **stat** zu verwenden, wie wir in Abschnitt 3.12 noch sehen werden. Falls die neue Datei kein Katalog ist, fahren wir fort und versuchen, sie zu entfernen. Falls dies schiefgeht, weil die Datei nicht existiert (**ENOENT**), so ist dies völlig in Ordnung, nicht jedoch, wenn es aus irgendeinem anderen Grund nicht funktioniert hat.

Diese Funktion klappt bei gewöhnlichen Dateien immer, bei Katalogen jedoch nur, wenn die effektive Benutzernummer Null (der Super-User) ist. Hätten wir gewollt, daß diese Funktion als Kommando zur allgemeinen Verfügung steht, so hätten wir den Super-User zum Besitzer der Datei gemacht und das *set-user-ID*-Bit gesetzt (**S_ISUID**, wie im vorigen Abschnitt erklärt). Allerdings sind wir zu feige, dies zu tun, da dann jedermann die Dateien und Kataloge anderer verlagern oder umbenennen könnte, wenn dieses Kommando mit den Rechten des Super-Users läuft![8]

Es gab eine Zeit, in der das echte UNIX-Kommando **mv** tatsächlich fast so funktionierte wie in unserem Beispiel. Wenn man einen Katalog verlagerte oder umbenannte, so führte man das Kommando wirklich als Super-User aus. Dies konnte

[8] Wir auch. Und Sie? A.d.Ü.

von schlauen Benutzern dazu mißbraucht werden, einzubrechen und Super-User zu werden:[9]

1. Kopieren Sie sich die Paßwortdatei (**/etc/passwd**) nach **/tmp**. Sie werden zum Besitzer der Kopie und haben außerdem Schreibrecht.
2. Editieren Sie **/tmp/passwd** und löschen Sie das Paßwort des Super-Users. (Der Login-Name des Super-Users ist **root**.)
3. Benennen Sie **/tmp** in **/etc** um:

```
mv /etc /etc2
mv /tmp /etc
```

4. Melden Sie sich als **root** an. Kein Paßwort wird benötigt.
5. Versetzen Sie so schnell Sie können alles in den ursprünglichen Zustand zurück.
6. Senden Sie Ihrem Systemadministrator einen Brief, der natürlich vom Benutzer **root** stammen wird.
7. *Melden Sie sich ab.* Schließlich wollen Sie ja nichts beschädigen, außer den Stolz des Systemadministrators.

Die Reparatur dieses Fehlers bestand darin, **mv** so zu erweitern, daß bei der Ausführung zur Bestimmung der Rechte die *wirkliche* Benutzernummer herangezogen wird. Wir wollen sichergehen, daß wir bei Ausführung dieses Programms als Super-User nur **link** und **unlink** auf Kataloge ausführen dürfen. Der Systemaufruf **access**, der im nächsten Abschnitt eingehend beschrieben wird, macht diesen Prüfvorgang ziemlich einfach.

Ein weiterer Fehler beim Anwenden von **mv** auf Kataloge existiert auch heute noch. Zwischen den Aufrufen von **link** und **unlink** besteht eine kleine Zeitspanne, während dieser es zwei Verweise auf den Katalog gibt. Wenn nun gleichzeitig ein Programm läuft, das davon ausgeht, daß die Kataloghierarchie ein vollkommener Baum ist, so kann es passieren, daß ein Katalog zweimal besucht wird. Die Möglichkeit hierfür ist gering, und das Risiko eines ernsten Schadens ist noch geringer. Doch um sicherzugehen, sollten Sie Programme, die die Katalogstruktur ablaufen, nur auf einem sonst unbenutzten System ausführen.

link kann Verweise von einem Katalog auf eine Datei oder auf einen anderen Katalog nur auf der gleichen logischen Platte erzeugen. Dies liegt daran, daß eine montierte logische Platte an einem einzigen Punkt im Dateisystem eingehängt ist. Anstatt hier näher darauf einzugehen, stellen wir die Erörterung montierbarer logischer Platten bis Abschnitt 9.14 zurück.

[9] Alan L. Glaser und ich gingen 1973 oder 1974 nach dieser Methode vor, um in das System von Dennis Ritchie und Ken Thompson einzubrechen.

3.7 Der Systemaufruf "access"

```
int access(path, pattern)  /* Zugriffsrechte pruefen */
    char * path;           /* zu untersuchende Datei */
    int pattern;           /* Zugriffsrechte-Bits    */

/* liefert 0 falls der verlangte Zugriff erlaubt ist */
/* oder -1 bei Fehler                                */
```

Im Gegensatz zu allen anderen Systemaufrufen, die sich mit Zugriffsrechten befassen, prüft **access** die *wirkliche* Benutzer- oder Gruppennummer, nicht die *effektive*. Das Argument **pattern** wird in Form von drei Zugriffsschutz-Bits verstanden: Lesen, Schreiben und Ausführen. Wenn die wirkliche Benutzernummer und die Benutzernummer des Besitzers von **path** übereinstimmen, so wird das Muster mit den Zugriffsrechten des Besitzers der Datei verglichen (Bit 8, 7 und 6 der Dateiattribute). Wenn die wirkliche Benutzernummer alle durch das Muster angegebenen Rechte besitzt, so liefert **access** 0 zurück, andernfalls –1. Wenn zwar nicht die wirklichen Benutzernummern, doch die wirklichen Gruppennummern übereinstimmen, so werden die Zugriffsrechte der Gruppe (Bit 5, 4 und 3) untersucht. Wenn auch die Gruppennummer nicht paßt, dann werden die Zugriffsrechte der Welt (Bit 2, 1 und 0) geprüft.

Wenn **pattern** Null ist, so untersucht **access** nur, ob **path** existiert. Jeder Katalog in **path** muß Ausführungsrecht (Suchrecht) besitzen.[10]

Wir können **access** dazu verwenden, das Sicherheitsloch im **mv**-Beispiel aus dem vorigen Abschnitt zu stopfen. Wir wollen sicherstellen, daß die wirkliche Benutzernummer Schreibrecht auf den Vater-Katalog der zu verlagernden Datei oder des zu verlagernden Katalogs und auf den neuen Vater-Katalog besitzt. Zur Unterstützung benötigen wir eine Funktion **parent**, die einen Pfad liest, der eine Datei oder einen Katalog angibt, und uns den entsprechenden Vater-Katalog liefert. Bei gegebenem Pfad **/usr/marc/book** zum Beispiel würde **parent** als Ergebnis **/usr/marc** liefern. Die Aufgabe ist einfach: Wir entfernen einfach das am weitesten rechts befindliche /-Zeichen und alles, was rechts davon liegt. Die einzige kleine Schwierigkeit besteht bei der korrekten Behandlung von Pfaden ohne /-Zeichen oder bei Pfaden, die nur ein führendes /-Zeichen besitzen. Hier ist unsere Version von **parent**:

```
char * parent(path, dir)        /* Vater-Katalog besorgen */
    char * path, * dir;
{
    char * lastslash, * strrchr();
```

[10] Und zwar für den *wirklichen* Benutzer. A.d.Ü.

```
    strcpy(dir, path);
    lastslash = strrchr(dir, '/');
    if (lastslash == NULL)      /* Arbeitskatalog        */
        strcpy(dir, ".");
    else if (lastslash == dir)  /* Wurzel                */
        strcpy(dir, "/");
    else
        * lastslash = '\0';
    return(dir);
}
```

Die Standardfunktion **strrchr** liefert einen Zeiger auf den Punkt, wo ein Zeichen letztmals in einer Zeichenkette vorkommt, oder **NULL**, falls es überhaupt nicht vorkommt.

Nun sind wir in der Lage, **mv** zu korrigieren, indem wir folgende Zeilen an den Anfang der Funktion setzen:

```
char dir[50], * parent();

if (access(parent(from, dir), 2) == -1
    || access(parent(to, dir), 2) == -1)
    syserr(dir);
```

Denken wir darüber nach, was unsere Aufgabe war. Da die Notwendigkeit bestand, **link** und **unlink** auf Kataloge anzuwenden, mußten wir das Programm als Super-User ausführen. Doch dies bedeutete, daß das Zugangssystem des Kerns völlig umgangen wurde, so daß es unmöglich war, Benutzer daran zu hindern, unerlaubte Dinge zu tun. Wir mußten unser eigenes Zugangssystem einrichten, indem wir **access** benützen. Dies verdeutlicht das zentrale Sicherheitsproblem in UNIX: Nicht, daß UNIX nicht sicher gemacht werden kann, sondern daß die Verantwortung so weit verstreut ist. Es gibt keine Programmiermethodik in UNIX, die Sicherheitslücken vermeidet, und auch keinen gangbaren Weg, ein UNIX-System auf Sicherheit zu prüfen. Die Schlupflöcher sind nach und nach über eine Reihe von Jahren hinweg entdeckt worden. Heutzutage ist ein UNIX-System wahrscheinlich ziemlich sicher, falls der Systemadministrator auf Draht ist. *Doch niemand weiß dies ganz sicher!*

3.8 Der Systemaufruf "mknod"

```
#include <sys/types.h>
#include <sys/stat.h>
```

```
int mknod(path, mode, device)  /* Inode erzeugen                    */
    char * path;               /* zu erzeugende Datei               */
    int mode;                  /* Dateiattribute der neuen Datei    */
    int device;                /* Geraetenummer                     */

/* liefert 0 bei Erfolg oder -1 bei Fehler                          */
```

mknod erzeugt eine Datei beliebigen Typs. Alle 16 Bits des zweiten Arguments **mode** werden benützt, wie in Abschnitt 3.5 erklärt. Dies ist die einzige Möglichkeit, Kataloge, Gerätedateien und FIFO-Dateien zu erzeugen. Gewöhnliche Dateien werden dagegen fast immer durch **creat** oder **open** erzeugt.

Erzeugt man eine Gerätedatei, so muß die Gerätenummer angegeben werden, die, wie in Abschnitt 1.2.3 erwähnt, einen Index in einen Vektor innerhalb des Kerns darstellt, und den passenden Gerätetreiber bestimmt. Die tatsächliche Deutung einer Gerätenummer ist abhängig von der Konfiguration, doch das am weitesten rechts stehende Byte ist im allgemeinen die *minor*-Gerätenummer, und das nächste Byte die *major*-Gerätenummer. Die *major*-Gerätenummer wählt den Gerätetreiber, während die *minor*-Gerätenummer das genaue physikalische Gerät angibt. Ein einzelner Gerätetreiber für eine Platte könnte zum Beispiel mehrere wirkliche Platten bedienen. Per Konvention werden alle Gerätedateien in den Katalog **/dev** gesteckt.

mknod wird nur dann zum Erzeugen einer Gerätedatei benutzt, wenn ein neuer Gerätetreiber installiert oder eine neue *minor*-Gerätenummer wirksam wird. Wenn ein UNIX-System einmal eingerichtet ist, ist **mknod** für Gerätedateien bald nicht mehr von großer Bedeutung.

Abgesehen von der Erzeugung einer FIFO-Datei kann nur der Super-User **mknod** ausführen. Bei Gerätedateien ist diese Beschränkung verständlich, doch bei Katalogen (und was das betrifft, auch bei gewöhnlichen Dateien) scheint sie ein Fehler zu sein.[11] Doch UNIX ist schon immer so gewesen, so daß die Programmierer viel Zeit hatten, sich daran zu gewöhnen.

Systemprogrammierer, die UNIX-Neulinge sind, suchen oft vergeblich nach dem Systemaufruf **mkdir**. Kennen Sie den Systemaufruf, den das **mkdir**-Kommando benützt? Es gibt keinen derartigen Systemaufruf, doch es ist leicht genug, eine Funktion zu schreiben, die die gleiche Arbeit erledigt:

```
int mkdir(path)  /* Katalog erzeugen; nur fuer Super-User */
    char * path;
{
    return(mknod(path, S_IFDIR | 0775, 0));
}
```

[11] Nein! Man verläßt sich darauf, daß . und .. am Anfang eines Katalogs stehen, denn sonst funktioniert z.B. das pwd-Kommando nicht. A.d.Ü.

Da nur der Super-User **mknod** ausführen darf, läuft das **mkdir**-Kommando einfach mit gesetztem *set-user-ID*-Bit. Doch auch gewöhnliche Benutzer schreiben oft Programme, die Kataloge erzeugen müssen. Sie können das **mkdir**-Kommando einfach als Sub-Prozeß ausführen. Hier ist eine Version, die von jedermann ausgeführt werden kann:

```
int mkdir2(path)  /* Katalog erzeugen */
    char * path;
{
    char cmd[50];

    sprintf(cmd, "mkdir %s", path);
    if (system(cmd) == 0)
        return(0);
    else
        return(-1);
}
```

Dies ist eine zeitraubende Methode, einen Katalog zu erzeugen, doch Kataloge werden nicht so oft erzeugt, daß das ins Gewicht fällt.

Als FIFO-Dateien zum ersten Mal eingeführt wurden, war es ein echtes Rätsel, wie man sie erzeugt. Der Begriff tauchte nicht einmal im *permutierten Index* des UNIX-Handbuchs auf! Die Programmierer entdeckten zufällig, daß für diese Aufgabe **mknod** zuständig war, wie durch diese handliche Funktion veranschaulicht wird:

```
int mkfifo(path)  /* FIFO-Datei erzeugen */
    char * path;
{
    return(mknod(path, S_IFIFO | 0666, 0));
}
```

In Kapitel 7 werden wir sehr viel mehr über FIFO-Dateien erzählen.

3.9 Der Systemaufruf "chmod"

```
int chmod(path, mode)  /* Zugriffsschutz einer Datei aendern */
    char * path;       /* zu modifizierende Datei            */
    int mode;          /* neuer Zugriffsschutz               */

/* liefert 0 bei Erfolg oder -1 bei Fehler                   */
```

Der Systemaufruf **chmod** ändert einen Teil der Dateiattribute einer existenten Datei beliebigen Typs. Er kann jedoch nicht dazu verwendet werden, den Typ selbst zu ändern, so daß nur die zwölf am weitesten rechts stehenden Bits von **mode** Verwendung finden. Die effektive Benutzernummer des Aufrufers muß mit der Benutzernummer des Dateibesitzers übereinstimmen, oder der Aufrufer muß der Super-User sein. Hier reicht es nicht, daß die effektive Gruppennummer mit der der Datei übereinstimmt. Nur der Super-User darf das *save-swapped-text*-Bit setzen.

Nebenbei bemerkt: Benutzen Sie niemals eine Attributkombination wie **04777**. Dadurch wird nämlich das *set-user-ID*-Bit gesetzt und allen anderen erlaubt, diese Datei zu schreiben. Ein anderer Benutzer könnte dann die Shell auf Ihre Datei kopieren, sie ausführen und damit (effektiv) Sie werden! Deshalb wird ein guter Systemadministrator von Zeit zu Zeit ein Programm laufen lassen, das nach Dateien Ausschau hält, die solche Attribute besitzen, und die unerwünschten Schreib-Bits löschen.

Ein Beispiel, in dem **chmod** verwendet wird, finden Sie in Abschnitt 3.12.

3.10 Der Systemaufruf "chown"

```
int chown(path, owner, group)  /* Besitzer einer Datei aendern  */
    char * path;               /* zu modifizierende Datei       */
    int owner;                 /* neue Besitzernummer           */
    int group;                 /* neue Gruppennummer            */

/* liefert 0 bei Erfolg oder -1 bei Fehler                      */
```

chown ändert bei einer Datei die Benutzer- und Gruppennummer des Besitzers. Nur der Besitzer oder der Super-User dürfen diesen Systemaufruf ausführen.

Solange der Aufrufer nicht der Super-User ist, löscht **chown** das *set-user-ID*-Bit und das *set-group-ID*-Bit. Dies dient der Unterbindung einer eher offensichtlichen Form des Einbruchs:

```
$ cp /bin/sh mysh     [erzeuge eine private Kopie der Shell]
$ chmod 4700 mysh     [setze das set-user-ID-Bit]
$ chown root mysh     [mache den Super-User zum Besitzer]
$ mysh                [werde Super-User]
```

Wenn Sie Ihre eigene Super-User-Shell besitzen möchten, müssen Sie die Reihenfolge von **chmod** und **chown** umkehren. Doch sofern Sie nicht bereits der Super-User sind, dürfen Sie dann **chmod** nicht ausführen. Damit ist ein weiteres Hintertürchen zugesperrt.

In Abschnitt 3.12 gibt es ein Beispiel zu **chown**.

3.11 Der Systemaufruf "utime"

```
#include <sys/types.h>

int utime(path, timep)  /* Zeitstempel von Dateien aendern  */
    char * path;        /* zu modifizierende Datei          */
    struct utimbuf
    {
        time_t atime;   /* neuer Zeitstempel fuer Zugriff   */
        time_t mtime;   /* neuer Zeitstempel fuer Aenderung */
    } * timep;

/* liefert 0 bei Erfolg oder -1 bei Fehler                  */
```

utime ändert die Zeitstempel für Zugriff und Änderung einer Datei beliebigen Typs. **time_t** steht in Wirklichkeit für eine long-Integer, und die Zeitangaben sind in normalem Kalenderzeitformat angegeben, wie in Abschnitt 3.4 definiert (Anzahl Sekunden seit 1970). Wenn das zweite Argument, **timep**, den Wert **NULL** hat, so werden die Zugriffs- und Änderungszeitstempel auf die momentane Zeit gesetzt. Der Sinn dieser Sache ist, daß eine Datei up-to-date erscheinen kann, ohne daß man sie selbst neu schreiben muß. Den größten Nutzen hieraus zieht das **touch**-Kommando.

Falls **timep** ungleich **NULL** ist, so kann nur der Besitzer oder der Super-User die Zeitstempel ändern. Ist **timep** dagegen **NULL**, dann kann auch jeder andere, der Schreibrecht auf die Datei hat, die Zeitstempel ändern. Dieses Zugeständnis wurde gemacht, damit das **touch**-Kommando sinnvoller wurde. Wenn Sie Schreibrecht auf eine Datei besitzen, sollten Sie auch in der Lage sein, ihre Zeitstempel zu erneuern.

Abgesehen von **touch** wird **utime** meistens dann benützt, wenn Dateien von einem Band restauriert werden oder wenn man Dateien über eine Datenübertragungsleitung erhält. Die Zeitstempel werden auf die Werte zurückgesetzt, die sie ursprünglich besaßen. Der Statusänderungszeitstempel kann nicht zurückgesetzt werden, was auch sinnvoll ist, da ja die Inode nicht verschoben, sondern eine neue geschaffen wurde.

Obwohl die Handbücher von System III und System V das Argument **timep** in Form einer Struktur beschreiben, gibt es in diesen UNIX-Versionen keine Definitionsdatei, die diese Definition beinhaltet. Dies zweifellos deswegen, da in Version 7 **timep** etwas anders definiert wurde:

```
time_t timep[2];
```

Allem Anschein nach beschlossen die Autoren des Handbuchs von System III, ein wenig nobler zu sein.

Im nächsten Abschnitt wird ein Beispiel gezeigt, in dem **utime** benützt wird.

3.12 Die Systemaufrufe "stat" und "fstat"

```
#include <sys/types.h>
#include <sys/stat.h>


int stat(path, sbuf)      /* Dateistatus besorgen */
    char * path;          /* Pfadname             */
    struct stat * sbuf;   /* Statusinformation    */

/* liefert 0 bei Erfolg oder -1 bei Fehler       */


int fstat(fd, sbuf)       /* Dateistatus besorgen */
    int fd;               /* Filedeskriptor       */
    struct stat * sbuf;   /* Statusinformation    */

/* liefert 0 bei Erfolg oder -1 bei Fehler       */
```

stat und **fstat** werden verwendet, um die Statusinformation aus einer Inode zu erhalten. **stat** benützt einen Pfad und findet die zugehörige Inode durch Verfolgen dieses Pfades; **fstat** verwendet einen aktiven Filedeskriptor und findet die zugehörige Inode aus der Tabelle der aktiven Inodes im Kern. Bei beiden Aufrufen werden die Daten aus der Inode neu geordnet und in der vom Benutzer zur Verfügung gestellten Struktur **stat** abgelegt, die in der Definitionsdatei **/usr/include/sys/stat.h** definiert ist.

fstat kann auch auf einen mit einer Pipe verbundenen Filedeskriptor angewandt werden, obwohl man normalerweise hinter einer Pipe keine Inode vermutet. Und doch ordnet der Kern tatsächlich jeder Pipe eine Inode zu, und **fstat** befragt diese verborgene Inode. Der Typ dieser Datei ist der einer FIFO-Datei. Da Pipes, im Gegensatz zu FIFO-Dateien, keine Namen besitzen, ist die Verweiszahl Null, und mit Hilfe dieser Tatsache lassen sich Pipes von FIFO-Dateien unterscheiden, falls dies jemals nötig sein sollte. Die Zugriffsschutz-Bits sind auf Null gesetzt. Die (Datei-)Größe gibt die Zahl der Bytes in der Pipe an, die darauf warten, gelesen zu werden. Die Einträge für Besitzer, Gruppe und Zeitstempel haben die gleiche Bedeutung wie bei Dateien.

Die folgende Tabelle zeigt die Komponenten der **stat**-Struktur. In Klammern angegeben sind der Typname, wie er in der Strukturdefinition erscheint, und der tatsächliche C-Typ, wie er in **/usr/include/sys/types.h** definiert ist. Sie müssen diesen kennen, wenn sie die Komponenten in einem Programm benützen wollen (**localtime** zum Beispiel benützt eine long-Integer, keine **time_t**-Variable). Der Typ **ushort** bedeutet **unsigned short**.

st_dev	Die Gerätenummer des Geräts, das die Inode enthält. In der üblichen Interpretation ist das am weitesten links liegende Byte die *major*-Nummer, und das am weitesten rechts liegende Byte die *minor*-Nummer. (**dev_t**; **short**)
st_ino	Die Inumber. (**ino_t**; **ushort**)
st_mode	Die 16-Bit-Dateiattribute (siehe Abschnitt 3.5). (**ushort**)
st_nlink	Die Verweiszahl; 0 bei Pipes. (**short**)
st_uid	Die Benutzernummer des Besitzers. (**ushort**)
st_gid	Die Gruppennummer des Besitzers. (**ushort**)
st_rdev	Bei Gerätedateien die Gerätenummer des entsprechenden Geräts. Bei anderen Dateitypen ist dieses Feld bedeutungslos. (**dev_t**; **short**)
st_size	Die augenblickliche Größe der Datei; 0 bei Gerätedateien. (**off_t**; **long**)
st_atime	Der Zugriffszeitstempel in Form von Sekunden seit 1970 (siehe Abschnitt 3.4). Dieser wird immer dann geändert, wenn die Datei gelesen wird, jedoch nicht, wenn ein Katalog in einem Pfad durchsucht wird. (**time_t**; **long**)
st_mtime	Der Änderungszeitstempel, der auf den neuesten Stand gebracht wird, wenn die Datei geschrieben wird oder wenn ein Verweis in einem Katalog hinzugefügt oder entfernt wird. (**time_t**; **long**)
st_ctime	Der Statusänderungszeitstempel, der geändert wird, wenn die Datei geschrieben wird, die Dateiattribute, der Besitzer, die Verweiszahl oder der Änderungszeitstempel geändert werden, und auch wenn der Zugriffszeitstempel explizit mittels **utime** geändert wird, nicht jedoch, wenn die Datei gelesen wird. (**time_t**; **long**)

Um zu demonstrieren, wie **stat** benützt wird, und auch um die bereits früher versprochenen Beispiele zu **chmod**, **chown** und **utime** zu zeigen, stellen wir **statutil** vor, ein interaktives Hilfsprogramm, das folgendes kann: Ausgabe von Statusinformation; Änderungen des Besitzers und der Dateiattribute, die den Zugriff bestimmen; Änderungen der Zeitstempel und sogar die Ausführung beliebiger Shell-Kommandos. Vielleicht finden Sie es nützlich, das Programm selbst auszuführen, um zu sehen, wie sich der Status ändert, wenn man verschiedene Operationen ausführt.

Hier ist ein Beispiel einer Sitzung mit **statutil**. Zeichen, die der Benutzer getippt hat, sind unterstrichen.

```
$ statutil
a  change access time
f  new file name
m  change modification time
o  change owner
p  change permissions
q  quit
s  display status
!  execute UNIX command
?  display command summary

Command? f

File? /dev/hd0

File "/dev/hd0"
Block special file
Device number: 0, 0
Resides on device: 0, 0
I-node: 93; Links: 1; Size: 0
Owner ID: 2; Name: bin
Group ID: 2; Name: bin
Permissions: 644
Last access:            Thu Aug 23 10:02:22 1984
Last modification:      Wed Dec 31 22:00:00 1969
Last status change:     Mon Aug 20 11:03:05 1984

Command? f

File? statutil

File "statutil"
Ordinary file
Resides on device: 0, 0
I-node: 698; Links: 1; Size: 22604
Owner ID: 200; Name: marc
Group ID: 1; Name: staff
Permissions: 755
Last access:            Thu Aug 23 11:49:44 1984
Last modification:      Thu Aug 23 11:49:39 1984
Last status change:     Thu Aug 23 11:49:40 1984
```

```
Command? p

Mode (up to 4 octal digits)? 6755

Command? s

File "statutil"
Ordinary file
Resides on device: 0, 0
I-node: 698; Links: 1; Size: 22604
Owner ID: 200; Name: marc
Group ID: 1; Name: staff
Set-user-ID
Set-group-ID
Permissions: 755
Last access:            Thu Aug 23 11:49:44 1984
Last modification:      Thu Aug 23 11:49:39 1984
Last status change:     Thu Aug 23 11:51:53 1984

Command? m

Time (YYMMDDhhmmss)? 781107085236

Command? s

File "statutil"
Ordinary file
Resides on device: 0, 0
I-node: 698; Links: 1; Size: 22604
Owner ID: 200; Name: marc
Group ID: 1; Name: staff
Set-user-ID
Set-group-ID
Permissions: 755
Last access:            Thu Aug 23 11:49:44 1984
Last modification:      Tue Nov  7 08:52:36 1978
Last status change:     Thu Aug 23 11:53:40 1984

Command? q
$
```

Bei der Programmierung von **statutil** verwendeten wir einige Standardfunktionen: Zum Zugriff auf die Paßwortdatei (**/etc/passwd**) und die Datei mit den Gruppennamen (**/etc/group**) zwecks Konvertierung von numerischen Angaben in Namen (und umgekehrt) benützten wir **getpwuid**, **getpwnam**, **getgrgid**, und **getgrnam**. Weiterhin verwendeten wir **asctime** und **localtime**, die wir schon in Abschnitt 3.4 benutzt hatten, und auch **timecvt**, das auch in jenem Abschnitt auftauchte.

Hier ist nun **statutil**. Sie sollten das Programm sorgfältig durchlesen, da es zu weit mehr fähig ist, als wir in der Sitzung zeigten.

```
#include <sys/types.h>
#include <sys/stat.h>
#include <pwd.h>
#include <grp.h>
#include <time.h>

main()  /* statutil */
{
    setbuf(stdout, NULL);
    help();
    mainloop();
}

static void mainloop()  /* Kommandos bearbeiten */
{
    char path[50], cmd[10], shcmd[100];

    while (1)
    {
        prompt("Command", cmd);
        if (strlen(cmd) > 1)
            cmd[0] = '\1';  /* Meldung 'unknown command' erzwingen */
        switch (cmd[0])
        {
            case '\0':
            case 'q':
                exit(0);
            case 'a':
            case 'm':
                chtime(path, cmd[0]);
                continue;
            case 'f':
                prompt("File", path);
```

```
                if (access(path, 0) == -1)
                {
                    printf("%s nonexistent\n", path);
                    continue;
                }
                status(path);
                continue;
            case 'o':
                chowner(path);
                continue;
            case 'p':
                chperms(path);
                continue;
            case 's':
                status(path);
                continue;
            case '!':
                prompt("Shell command", shcmd);
                system(shcmd);
                continue;
            case '?':
                help();
                continue;
            default:
                printf("Unknown command; use ? for help");
                continue;
        }
    }
}

static void help()  /* Auswahl zeigen */
{
    printf("a  change access time\n");
    printf("f  new file name\n");
    printf("m  change modification time\n");
    printf("o  change owner\n");
    printf("p  change permissions\n");
    printf("q  quit\n");
    printf("s  display status\n");
    printf("!  execute UNIX command\n");
    printf("?  display command summary\n");
}
```

```
static void prompt(msg, result)  /* Eingabeaufforderung */
    char * msg, * result;
{
    printf("\n%s? ", msg);
    if (gets(result) == NULL)
        exit(0);
}

static void status(path)  /* "s"-Kommando */
    char * path;
{
    struct stat sb;

    if (stat(path, &sb) == -1)
    {
        syserrmsg("stat");
        return;
    }
    printf("\nFile \"%s\"\n", path);
    dspstatus(&sb);
}

static void dspstatus(sbp)  /* Status zeigen */
    struct stat * sbp;
{
    BOOLEAN isdevice = FALSE;
    struct passwd * pw, * getpwuid();
    struct group * gr, * getgrgid();
    char * name, * asctime();
    struct tm * localtime();

    if ((sbp->st_mode & S_IFMT) == S_IFDIR)
        printf("Directory\n");
    else if ((sbp->st_mode & S_IFMT) == S_IFBLK)
    {
        printf("Block special file\n");
        isdevice = TRUE;
    }
```

```
    else if ((sbp->st_mode & S_IFMT) == S_IFCHR)
    {
        printf("Character special file\n");
        isdevice = TRUE;
    }
    else if ((sbp->st_mode & S_IFMT) == S_IFREG)
        printf("Ordinary file\n");
    else if ((sbp->st_mode & S_IFMT) == S_IFIFO)
        printf("FIFO\n");
    if (isdevice)
        printf("Device number: %d, %d\n",
              (sbp->st_rdev >> 8) & 0377, sbp->st_rdev & 0377);
    printf("Resides on device: %d, %d\n",
          (sbp->st_dev >> 8) & 0377, sbp->st_dev & 0377);
    printf("I-node: %d; Links: %d; Size: %ld\n",
           sbp->st_ino, sbp->st_nlink, sbp->st_size);
    if ((pw = getpwuid(sbp->st_uid)) == NULL)
        name = "???";
    else
        name = pw->pw_name;
    printf("Owner ID: %d; Name: %s\n", sbp->st_uid, name);
    if ((gr = getgrgid(sbp->st_gid)) == NULL)
        name = "???";
    else
        name = gr->gr_name;
    printf("Group ID: %d; Name: %s\n", sbp->st_gid, name);
    if ((sbp->st_mode & S_ISUID) == S_ISUID)
        printf("Set-user-ID\n");
    if ((sbp->st_mode & S_ISGID) == S_ISGID)
        printf("Set-group-ID\n");
    if ((sbp->st_mode & S_ISVTX) == S_ISVTX)
        printf("Save swapped text after use\n");
    printf("Permissions: %o\n", sbp->st_mode & 0777);
    printf("Last access:    %s", asctime(localtime(&sbp->st_atime)));
    printf("Last modification:  %s",
            asctime(localtime(&sbp->st_mtime)));
    printf("Last status change: %s",
            asctime(localtime(&sbp->st_ctime)));
}
```

```
static void chtime(path, which)  /* "a"- und "m"- Kommandos */
    char * path, which;
{
    char atime[20];
    long seconds, timecvt();
    struct stat sb;
    struct utimbuf
    {
        time_t actime;
        time_t modtime;
    } tb;

    if (stat(path, &sb) == -1)
    {
        syserrmsg("stat");
        return;
    }
    prompt("Time (YYMMDDhhmmss)", atime);
    if ((seconds = timecvt(atime)) <= 0)
        return;
    switch (which)
    {
        case 'a':
            tb.actime = seconds;
            tb.modtime = sb.st_mtime;
            break;
        case 'm':
            tb.actime = sb.st_atime;
            tb.modtime = seconds;
    }
    if (utime(path, &tb) == -1)
        syserrmsg("utime");
}

static void chowner(path)  /* "o"-Kommando */
    char * path;
{
    char oname[20], gname[20];
    int owner, group;
    struct passwd * pw, * getpwnam();
    struct group * gr, * getgrnam();
```

```
    prompt("Owner name", oname);
    if ((pw = getpwnam(oname)) == NULL)
    {
        printf("Unknown name\n");
        return;
    }
    owner = pw->pw_uid;
    prompt("Group name", gname);
    if ((gr = getgrnam(gname)) == NULL)
    {
        printf("Unknown name\n");
        return;
    }
    group = gr->gr_gid;
    if (chown(path, owner, group) == -1)
        syserrmsg("chown");
}

static void chperms(path)  /* "p"-Kommando */
    char * path;
{
    char smode[20];
    int mode;

    prompt("Mode (up to 4 octal digits)", smode);
    if (sscanf(smode, "%o", &mode) != 1)
    {
        printf("Invalid mode\n");
        return;
    }
    if (chmod(path, mode) == -1)
        syserrmsg("chmod");
}

static void syserrmsg(msg)  /* Fehlermeldung ausgeben */
    char * msg;
{
    extern int errno, sys_nerr;
    extern char * sys_errlist[];
```

```
    fprintf(stderr, "ERROR: %s (%d", msg, errno);
    if (errno > 0 && errno <= sys_nerr)
        fprintf(stderr, "; %s)\n", sys_errlist[errno]);
    else
        fprintf(stderr, ")\n");
}
```

Die Funktion **syserrmsg** ist einfach eine nichttödliche Form von **syserr**, da wir bei interaktiven Programmen im allgemeinen lieber auf die Kommandoebene zurückkehren möchten, als ganz aufzuhören.

3.13 Der Systemaufruf "fcntl"

```
#include <fcntl.h>

int fcntl(fd, cmd, arg)                    /* Datei manipulieren */
    int fd;                                /* Filedeskriptor     */
    int cmd;                               /* Kommando           */
    int arg;                               /* Argument           */

/* liefert einen Wert, einen Filedeskriptor oder -1 bei Fehler */
```

Im Gegensatz zu vielen Systemaufrufen in diesem Kapitel, die die Eigenschaften einer Datei verändern (wie **chmod** und **utime**), ändert **fcntl** die Eigenschaften eines aktiven *Filedeskriptors*. Fünf Befehle stehen dafür zur Verfügung, und die entsprechenden Symbole sind in der Definitionsdatei **/usr/include/fcntl.h** (die gleiche Definitionsdatei wie bei **open**) definiert:

F_DUPFD Liefern eines neuen Filedeskriptors, der eine Kopie des ersten Arguments **fd** ist. Diese Leistung wird zusammen mit dem Systemaufruf **dup** im Abschnitt 6.3 erklärt.

F_GETFD Liefern der *close-on-exec*-Flagge, die angibt, ob ein Filedeskriptor bei Aufruf des Systemaufrufs **exec** geschlossen wird. *close-on-exec* wird in Abschnitt 5.3 genauer untersucht.

F_SETFD Setzen der *close-on-exec*-Flagge.

F_GETFL Liefern der Flaggen, die den Status der Datei angeben. Dieser ist nicht zu verwechseln mit dem im vorigen Abschnitt beschriebenen Dateistatus. Die Flaggen entsprechen dem zweiten Argument von **open** (Abschnitt 2.6). Ist ein bestimmter Filedeskriptor gegeben, so läßt sich herausfinden, ob er nur zum Schreiben (**O_WRONLY**), zum Lesen und zum Schreiben (**O_RDWR**) usw. geöffnet ist.

F_SETFL Setzen der obigen Flaggen auf den Wert des dritten Arguments. Die Flaggen **O_RDONLY**, **O_WRONLY**, und **O_RDWR** können nicht verändert werden; sie können nur mit Hilfe von **open** oder, im Falle von **O_WRONLY**, mittels **creat** gesetzt werden. Nur **O_NDELAY** und **O_APPEND** können durch **fcntl** gesetzt werden.

Bis hierher sind die einzigen Funktionen von **fcntl**, die sich auf bereits Besprochenes beziehen, das Beschaffen der Flaggen, die den Status der Datei angeben, und das Umschalten der **O_APPEND**-Flagge. Wir werden auf **fcntl** zurückkommen, sobald wir mehr Verwendung dafür haben, was im nächsten Kapitel der Fall sein wird.

Als Beispiel zeigen wir hier eine Funktion, die die **O_APPEND**-Flagge setzt. Beachten Sie, daß wir uns zuerst den Status der Datei beschaffen müssen, da wir keine anderen Flaggen in Unordnung bringen wollen:

```
int setappend(fd)  /* O_APPEND-Flagge setzen */
    int fd;
{
    int flags;

    if ((flags = fcntl(fd, F_GETFL, 0)) == -1)
        syserr("fcntl");
    return(fcntl(fd, F_SETFL, flags | O_APPEND));
}
```

Hätten wir statt dessen die **O_APPEND**-Flagge *ausschalten* wollen, dann hätte der Ausdruck zum Setzen der Flaggen wie folgt lauten müssen:

```
flags & ~O_APPEND
```

3.14 Portabilität

Soweit es die in diesem Kapitel eingeführten Systemaufrufe betrifft, gibt es keine Unterschiede zwischen System III und System V. Es bestehen jedoch einige Unterschiede zu Version 7, darunter die folgenden nichttrivialen:

1. **fcntl** ist bei Version 7 überhaupt nicht vorhanden, wie man erwarten konnte, da die Form von **open** mit drei Argumenten ebenfalls fehlt. Die Fähigkeiten von **F_DUPFD** werden viel besser durch **dup** und **dup2** erledigt. (**dup2** wurde in System III und System V abgeschafft.) Das **F_SETFD**-Kommando zum Setzen oder Löschen der *close-on-exec*-Flagge wird in Version 7 durch die dortige Form von **ioctl** übernommen (siehe Abschnitt 4.4). In Version 7 gibt es keine

Möglichkeit zum Abfragen der *close-on-exec*-Flagge. Dies stellt allerdings keinen ernsthaften Mangel dar, da eine solche Abfrage keinem sinnvollen Zweck dient.

2. Wird **fstat** auf eine Pipe angewandt, so zeigt dies eine gewöhnliche Datei anstatt einer FIFO-Datei an, die es ja in Version 7 nicht gibt. In Version 7 sind einige zusätzliche Dateitypen für *multiplexed*-Dateien definiert, die in System III und System V nicht existieren.

3. **utime** akzeptiert **NULL** nicht als zweites Argument zum Setzen der Zugriffs- und Änderungszeitstempel auf den aktuellen Zeitpunkt. Das **touch**-Kommando in Version 7 liest und schreibt ein Byte einer Datei, um den gleichen Zweck zu erfüllen (dies scheint mir ausreichend zu sein).

In Berkeley 4.2 BSD ist **fcntl** implementiert, doch aus irgendeinem Grund besitzen die für **F_SETFD** und **F_GETFD** verwendeten Symbole für die Statusflaggen andere Namen als die bei **open**. **stat** ist im allgemeinen kompatibel, doch wurden zusätzliche Strukturkomponenten zur Unterstützung spezifischer Merkmale von 4.2 BSD dazugefügt. FIFO-Dateien sind nicht vorhanden, statt dessen sogenannte *socket*-Dateien, die ungefähr tausendmal einfallsreicher und viel schwieriger zu benützen sind. **utime** gibt es als Unterprogramm; der Systemaufruf heißt **utimes**. Interessanterweise wurde ein *Systemaufruf* **mkdir** hinzugefügt, und man muß nicht der Super-User sein, um ihn aufrufen zu können.

Wie üblich lautet der beste Rat für Portabilität, nur gemeinsame Aufrufe zu benutzen, wenn irgend möglich. Bei der Anwendung von **stat** interessiert der Dateityp selten, und die *close-on-exec*-Flagge kann normalerweise in Ruhe gelassen werden. Auch **fcntl** tut nicht übermäßig viel bei Nicht-Special-Files (Special-Files sind ohnehin völlig nichtportabel, wie wir im nächsten Kapitel sehen werden). Somit können jedenfalls die meisten Programme ohne Verwendung der Dinge geschrieben werden, die sich von Version zu Version unterscheiden.

Sollten Sie jedoch gezwungen sein, eine nichtportable Fähigkeit zu verwenden, so können Sie auf eine Methode zurückgreifen, die eine einheitliche Schnittstelle für eine trotzdem vielfältige Implementierung bietet. Der C-Präprozessor hilft dabei, wie in der folgenden Funktion zum Umschalten der *close-on-exec*-Flagge veranschaulicht wird:

```
#ifdef V7
#include <sgtty.h>
#else
#include <fcntl.h>
#endif
```

```
int clex(fd, on)  /* close-on-exec-Flagge an- bzw. ausschalten */
    int fd;
    BOOLEAN on;
{
#ifdef V7
    return(ioctl(fd, on ? FIOCLEX : FIONCLEX, NULL));
#else
    return(fcntl(fd, F_SETFD, on ? 1 : 0));
#endif
}
```

Aufgaben

3.1 Implementieren Sie das **pwd**-Kommando (es zeigt den vollen Pfadnamen des Arbeitskatalogs). Sie können voraussetzen, daß nur das Wurzel-Dateisystem montiert ist (siehe Abschnitt 9.14).

3.2 Schreiben Sie ein Programm zum Kopieren eines ganzen Baums von Katalogen und Dateien. Es sollte zwei Argumente besitzen: die Wurzel des Baums (z.B. **/usr/marc/book**) und die Wurzel der Kopie (z.B. **/usr/marc/backup/book**).

3.3 Schreiben Sie ein Programm, das die ganze Information ausgibt, die im Super-Block eines Dateisystems enthalten ist. Zum Testen Ihres Programms muß Ihnen der Systemadministrator Leserecht auf die Gerätedatei geben.

3.4 Schreiben Sie ein Programm, das die Zahl der Tage zwischen zwei Datumsangaben berechnet.

3.5 Schreiben Sie **isdir** aus Abschnitt 3.6 neu und benützen Sie dabei **stat** anstelle von **open**.

3.6 Schreiben Sie **access** als Funktion. Sie dürfen dazu jeden Systemaufruf außer **access** verwenden.

3.7 Implementieren Sie das **ls**-Kommando.

Kapitel 4: Ein- und Ausgabe für Terminals

4.1 Einführung

Ein- und Ausgabe für Terminals ist so kompliziert, daß wir dafür beinahe ein ganzes Kapitel benötigen. Bei normaler Ein- und Ausgabe für Terminals, mit der wir auch beginnen werden, gibt es keine Schwierigkeiten — sie ist sogar einfacher als Ein- und Ausgabe für Dateien. Was die Dinge so kompliziert macht, sind die zahlreichen Variationen beim **ioctl**-Systemaufruf.

Standardmäßig behandelt die Ein- und Ausgabe unter UNIX ein Terminal wie einen altmodischen, mechanischen Fernschreiber, jenes Gerät mit den schweren Typenhebeln, die beim Hochschnellen einen furchtbaren Krach machen. Datensichtgeräte, Funktionstasten, Mäuse und andere Geräte mit ähnlichem Zweck sowie sogenannte *bit-mapped*-Bildschirme werden nicht unterstützt. Einige dieser modernen Geräte, wie z.B. Datensichtgeräte und Funktionstasten, können durch Unterprogramme bearbeitet werden, die auf Standard-Gerätetreiber zurückgreifen — **curses** ist das bekannteste dieser Pakete. Andere Geräte, wie *bit-mapped*-Bildschirme, benötigen kundenspezifische Gerätetreiber, die bislang allerdings noch nicht den Weg in das Standard-UNIX von AT&T gefunden haben.

Da der Gegenstand dieses Kapitels ein Gerätetreiber ist, und nicht ein Bestandteil des Kerns selbst wie das Dateisystem, unterscheiden sich spezielle Merkmale von einer UNIX-Version zur anderen sogar noch mehr als gewöhnlich. Manchmal werden selbst an einzelnen UNIX-Anlagen Änderungen am Terminaltreiber vorgenommen. Denken Sie auch daran, daß nicht alle Merkmale der Ein- und Ausgabe für Terminals durch UNIX selbst bewirkt werden. Mit zunehmender Verwendung von programmierbaren Terminals, lokalen Netzwerken und Datenübertragungsvorrechnern ist der Zeichenfluß vielfältigen Bearbeitungen ausgesetzt, bevor UNIX ihn zu sehen bekommt und nachdem er UNIX wieder verlassen hat. Die Einzelheiten dieser Vor- und Nachbearbeitung unterscheiden sich viel zu sehr, als daß sie hier aufgeführt werden können. Wie üblich werden wir uns auf die Standard-Gerätetreiber konzen-

trieren, die mit System III und System V zur Verfügung gestellt werden. Mit diesen Informationen als Hintergrund sollten Sie in der Lage sein, Ihr eigenes System mit Hilfe des Handbuchs zu verstehen.

Außerdem werden wir in diesem Kapitel noch kurz Geräte streifen, die keine Platten oder Terminals sind.

4.2 Normale Ein- und Ausgabe für Terminals

Wir beginnen mit der Erklärung normaler Ein- und Ausgabe für Terminals direkt nach dem ersten Anmelden, d.h. bevor Sie das **stty**-Kommando benutzen, um diese Ihrem Geschmack anzupassen. In den nachfolgenden Abschnitten werden wir dann zeigen, wie man mit Hilfe der Systemaufrufe **fcntl** und **ioctl** die normale Funktionsweise verändert.

Es gibt dreierlei Wege, auf ein Terminal zur Ein- oder Ausgabe zuzugreifen:

1. Sie können die Character-Gerätedatei **/dev/tty** zum Lesen, zum Schreiben oder, was am häufigsten der Fall ist, sowohl zum Lesen als auch zum Schreiben öffnen. Diese Gerätedatei steht für das Kontrollterminal eines Prozesses (siehe Abschnitt 1.5).

2. Falls Sie den tatsächlichen Namen der Gerätedatei kennen (z.B. **/dev/tty04**), können Sie statt dessen diese öffnen. Wenn Sie nur das Kontrollterminal benötigen, stellt dies keinen Vorteil gegenüber der Verwendung des allgemeinen Namens **/dev/tty** dar. Diese Technik wird hauptsächlich benützt, um andere Terminals als das Kontrollterminal anzusprechen.

3. Nach Konvention erbt jeder Prozeß drei aktive Filedeskriptoren: die Standard-Eingabe, die Standard-Ausgabe und die Diagnose-Ausgabe. Diese können, brauchen jedoch nicht, mit dem Kontrollterminal verbunden sein. Normalerweise sollten sie auf jeden Fall verwendet werden, auch wenn sie statt dessen mit einer Datei oder einer Pipe verbunden sind, denn dann hat der Benutzer beschlossen, die Ein- oder Ausgabe umzulenken.

Wenn unbedingt auf das Terminal selbst zugegriffen werden muß, so sollte man eine Gerätedatei öffnen; dies würde man etwa bei einem bildschirmorientierten Editor oder bei einer dringlichen Fehlermeldung machen. Ansonsten benützt man am besten die bereits aktiven Filedeskriptoren. Allerdings sollten Sie darauf achten, daß Sie im Ein- und Ausgabe-Bereich geräteunabhängig programmieren, da Sie im voraus nicht genau wissen, ob Sie es nachher mit einem Terminal, einer Datei oder einer Pipe zu tun haben.

Die grundlegenden Systemaufrufe für die Ein- und Ausgabe für Terminals sind **open**, **read**, **write** und **close**.

Um ein Terminal zum Schreiben zu öffnen kann auch **creat** angewendet werden. Da hierzu jedoch die Terminal-Gerätedatei bereits existieren muß, und da Sie vom Terminal normalerweise auch lesen wollen, ist **open** beinahe immer besser. **lseek** ist bei Terminals wirkungslos.

Der Systemaufruf **read** verhält sich bei Terminals anders als bei Dateien: Er liefert nie mehr als eine Eingabezeile; bevor die ganze Zeile fertig ist, werden keine Zeichen geliefert, selbst wenn **read** nur ein einzelnes Zeichen verlangen sollte. Dies liegt daran, daß wir erst dann von der endgültigen Form der Zeile ausgehen können, wenn der Benutzer diese abgeschlossen hat — mit den Tasten *Erase* (#) oder *Kill* (@) könnte die Zeile sonst noch geändert oder sogar gelöscht werden. Die von **read** gelieferte Zahl wird dazu verwendet, die Anzahl der tatsächlich gelesenen Zeichen zu bestimmen.

Der Benutzer kann eine Zeile auf zwei verschiedene Arten abschließen. Am häufigsten beendet das *Newline*-Zeichen eine Zeile. Zwar wird in Wirklichkeit das *Return*-Zeichen (oktal 15) gedrückt, doch der Gerätetreiber wandelt dies in ein *Newline*-Zeichen (oktal 12) um. Alternativ kann der Benutzer durch Drücken von *Control*-D ein *EOT*-Zeichen (*end-of-transmission* — Übertragungsende) erzeugen. In diesem Fall steht die Zeile so wie sie ist, ohne abschließendes *Newline*, für **read** zur Verfügung. Ein besonderer Fall ist wichtig: Erzeugt der Benutzer *EOT* am Zeilen*anfang*, so liefert **read** Null zurück, da die durch *EOT* beendete Zeile leer ist. Dies sieht nach Dateiende aus, und deshalb kann man sich *Control*-D als Dateiende-'Zeichen' vorstellen.

Die folgende Funktion kann dazu verwendet werden, mit Filedeskriptor 0 von einem Terminal zu lesen. Sie entfernt ein eventuell vorhandenes abschließendes *Newline* und fügt ein abschließendes Nullbyte an, um die Zeile in eine C-Zeichenkette zu verwandeln.

```
BOOLEAN getln(s, max)  /* Zeile vom Terminal lesen */
    char * s;
    int max;
{
    int nread;

    switch (nread = read(0, s, max - 1))
    {
        case -1:
            syserr("read");
        case 0:
            return(FALSE);
        default:
            if (s[nread - 1] == '\n')
                nread--;
```

```
            s[nread] = '\0';
            return(TRUE);
        }
    }
```

Bei Terminals ist diese Funktion effizient, da sie mit einem einzigen Systemaufruf die ganze Zeile liest. Außerdem braucht sie nicht nach dem Zeilenende zu suchen — sie richtet sich einfach nach der von **read** gelieferten Zahl. Dies funktioniert bei Dateien oder Pipes nicht — da die Beschränkung auf eine Zeile nicht zutrifft, liest **read** im allgemeinen zuviel. Statt einer Zeile würde **getln** die nächsten **max**–1 Zeichen lesen.

Eine allgemeinere Fassung von **getln** würde das besondere Verhalten von Terminals, höchstens eine Zeile zu lesen, ignorieren, und einfach jedes Zeichen mit *Newline* vergleichen:

```
BOOLEAN getln2(s, max)  /* Zeile aus beliebiger Eingabequelle lesen */
    char * s;
    int max;
{
    int i;
    char c;

    i = 0;
    while (TRUE)
        switch (read(0, &c, 1))
        {
            case -1:
                syserr("read");
            case 0:
                return(FALSE);
            default:
                if (i >= max)
                    fatal("getln2 overflow");
                if (c == '\n')
                {
                    s[i] = '\0';
                    return(TRUE);
                }
                s[i++] = c;
        }
}
```

Diese Version behandelt ein getipptes *EOT*-Zeichen, das an beliebiger Stelle außer am Zeilenanfang steht, anders als die frühere Version. Die Zeichenkette **s** wird erst dann korrekt durch ein Nullbyte abgeschlossen, wenn ein *Newline*-Zeichen gelesen wird. Da man nicht beabsichtigt, ihren Inhalt zu verwenden, wenn **getln2** als Resultat **FALSE** liefert, wird eine durch *EOT* beendete Zeile gänzlich übersprungen. In manchen Anwendungen mag dies sogar ein Vorteil sein. Falls es doch ein Nachteil sein sollte, so kann dies leicht repariert werden, indem man bei der Behandlung des Dateiendes (**case 0**) ein wenig Programmtext anfügt, in dem geprüft wird, ob irgendwelche Zeichen zu **s** hinzugekommen sind. Sollte dies der Fall sein, so wird **s** abgeschlossen und **TRUE** zurückgeliefert. Weiterhin wird eine Flagge gesetzt, so daß der nächste Aufruf von **getln2** sofort **FALSE** liefert.

getln2 liest zwar Terminals, Dateien und Pipes korrekt, jedoch werden Dateien und Pipes nicht mit optimaler Geschwindigkeit gelesen, da die Eingabe nicht, wie in Abschnitt 2.10 beschrieben, gepuffert wird. Dies kann leicht korrigiert werden, indem man **getln2** so ändert, daß es **Sgetc** aufruft, das Teil des in Abschnitt 2.10 eingeführten **STREAM**-Pakets ist. **Sopen** ist bereits zum Öffnen von Terminal-Gerätedateien (wie z.B. **/dev/tty**) geeignet. Um jedoch **Sgetc** auf die Standard-Eingabe anwenden zu können, müssen wir eine Funktion namens **Sfdopen** hinzufügen, die einen **STREAM**-Zeiger mittels eines bereits aktiven Filedeskriptors anstelle eines Pfads initialisiert. Dann sind wir in der Lage, ein Zeichen von der Standard-Eingabe zu lesen, egal, ob sich dahinter ein Terminal, eine Datei oder eine Pipe verbirgt:

```
STREAM * stin;
char c;

if ((stin = Sfdopen(0, "r")) == NULL)
    syserr("Sfdopen");
c = Sgetc(stin);
```

Nun lesen wir in jedem Fall so schnell wie möglich: blockweise bei Dateien und Pipes, und zeilenweise bei Terminals. Der bereits früher erwähnte Fehler der 'verlorengegangenen Zeile' ist nicht im **STREAM**-Paket selbst begründet. Ob wir eine durch *EOT* beendete Zeile verlieren, hängt davon ab, wie wir die Funktion schreiben, die **Sgetc** aufruft. Die für **getln2** vorgeschlagene Lösung würde auch in diesem Fall funktionieren.

Unsere Implementierung des **STREAM**-Pakets erlaubt es nicht, den gleichen **STREAM**-Zeiger sowohl zur Eingabe als auch zur Ausgabe zu verwenden. Falls also tatsächlich Ausgabe auf das Terminal verlangt wird, so muß unter Verwendung von Filedeskriptor 1 ein zweiter **STREAM** geöffnet werden.

Die Standard-Bücherei der Ein- und Ausgabe unter UNIX stellt drei vordefinierte **FILE**-Zeiger zum Zugriff auf das Terminal zur Verfügung: **stdin**, **stdout**

und **stderr**. Deshalb muß die Funktion **fdopen**, die mit unserer Funktion **Sfdopen** vergleichbar ist, im allgemeinen nicht aufgerufen werden.

Ausgabe auf ein Terminal ist einfacher als Eingabe, da keine Bearbeitung von *Erase*- oder *Kill*-Zeichen stattfindet. Exakt so viele Zeichen, wie wir mit **write** ausgeben, werden sofort an das Terminal geschickt, gleichgültig, ob ein *Newline*-Zeichen vorhanden ist oder nicht. Dies ist tatsächlich wesentlich, da wir sonst nicht in der Lage wären, Dinge zu tun wie die Ausgabe eines Prompts ohne Wagenrücklauf an den Benutzer, wie es beim **statutil**-Programm in Abschnitt 3.12 der Fall war.

Wird **close** auf einen Filedeskriptor angewandt, der mit einem Terminal verbunden ist, so hat dies den gleichen Effekt wie bei Dateien: Der Filedeskriptor wird einfach zur Wiederverwendung zur Verfügung gestellt. Da der Filedeskriptor meistens 0, 1 oder 2 ist, ist eine Verwendung nicht sofort offensichtlich. Deshalb hält sich niemand damit auf, die Filedeskriptoren am Programmende zu schließen.[1]

4.3 Blockierungsfreie Ein- und Ausgabe für Terminals

Wenn bei der Ausführung von **read** auf ein Terminal keine Zeile mit Daten verfügbar ist, wartet **read**. Da der Prozeß in der Zwischenzeit nichts tun kann, wird dies *Blockierung* (*blocking*) genannt. Bei Dateien gibt es keine vergleichbare Situation: entweder sind Daten verfügbar oder das Dateiende ist erreicht.

Die **O_NDELAY**-Flagge (*no delay*: verzögerungsfrei), die durch **open** oder **fcntl** gesetzt wird, verhindert die Blockierung bei **read**. Falls dann keine Zeile mit Daten verfügbar ist, kehrt **read** sofort mit dem Resultat Null zurück. Dies ist nicht zu unterscheiden von einem Dateiende, das erzeugt wird, wenn der Benutzer ein *EOT*-Zeichen tippt. Diese Mehrdeutigkeit muß immer berücksichtigt werden, wenn blockierungsfreie Ein- und Ausgabe für Terminals verwendet wird. Die einfachste Lösung besteht darin, auf *EOT* (*Control*-D) zu verzichten und statt dessen eine Alternative wie **q** (für '*quit*') oder *Control*-X zu verwenden.

Der vorgeschlagene Standard von */usr/group* hat dieses Problem dadurch beseitigt, indem er **read** bei Nichtvorliegen von Daten –1 zurückliefern läßt und **errno** auf den Wert **EAGAIN** setzt. Doch bisher hat keine der wichtigeren UNIX-Versionen diesen Vorschlag übernommen. Und selbst wenn dies bei einer zukünftigen UNIX-Version der Fall sein sollte, so dürfen wir annehmen, daß es noch viele Jahre lang andere Versionen geben wird, die nach der alten Art und Weise arbeiten.

Oftmals wollen wir die Blockierung nach Belieben an- und abschalten, so daß wir eine Funktion **setblock** schreiben, die **fcntl** geeignet aufruft. Es genügt nicht, wenn wir **fcntl** einfach mit **O_NDELAY** als Argument aufrufen, da wir aufpassen müssen, daß die anderen Flaggen unversehrt bleiben. Es ist zwar richtig, daß es

[1] In Kapitel 6 schließen wir sie, wenn wir zwei Prozesse mit einer Pipe verbinden.

heutzutage keine weiteren Flaggen gibt, die auf Terminals wirken, da **O_APPEND** die einzige weitere Flagge ist, die durch **fcntl** gesetzt werden kann. Dennoch ist es eine gute Idee, unsere Programme gegen zukünftige Störungen immun zu machen, soweit dies möglich ist.

```
void setblock(fd, on)  /* Ein- und Ausschalten                    */
    int fd;            /* blockierungsfreier Eingabe              */
    BOOLEAN on;
{
    static int blockf, nonblockf;
    static BOOLEAN first = TRUE;
    int flags;

    if (first)
    {
        first = FALSE;
        if ((flags = fcntl(fd, F_GETFL, 0)) == -1)
            syserr("fcntl");
                    /* sicherstellen, dass O_NDELAY aus ist */
        blockf = flags & ~O_NDELAY;
                    /* sicherstellen, dass O_NDELAY an ist  */
        nonblockf = flags | O_NDELAY;
    }
    if (fcntl(fd, F_SETFL, on ? blockf : nonblockf) == -1)
            syserr("fcntl2");
}
```

Man kann die **O_NDELAY**-Flagge auch dann setzen, wenn man mit **open** eine Terminal-Gerätedatei öffnet. In diesem Fall beeinflußt **O_NDELAY** den Systemaufruf **open** ebenso wie **read**: Falls keine Verbindung zustandekommt (*no carrier*), kehrt **open** zurück, ohne auf eine zu warten.

Eine Anwendung blockierungsfreier Eingabe besteht in der Überwachung mehrerer Terminals. Die Terminals können z.B. Laborgeräte sein, die durch Terminalanschlüsse mit einem UNIX-Rechner verbunden sind. Gelegentlich werden Zeichen hereingeschickt, die wir so ansammeln wollen, wie sie hereinkommen, unabhängig von der Reihenfolge, in der sie auftauchen. Da wir nicht voraussagen können, wann ein bestimmtes Terminal ein Zeichen überträgt, können wir blockierende Ein- und Ausgabe nicht gebrauchen, da wir sonst vielleicht auf ein Terminal warten, das nichts zu sagen hat, während andere gesprächige Terminals ignoriert würden. Mit Hilfe von blockierungsfreier Ein- und Ausgabe sind wir jedoch in der Lage, alle Terminals der Reihe nach abzufragen; wenn ein Terminal nicht bereit ist, liefert **read** Null und wir machen einfach weiter. Wenn wir eine komplette Runde beendet

haben, ohne irgendwelche Daten zu finden, so warten wir eine Sekunde, bevor wir wieder loslegen, um nicht die ganze CPU an uns zu reißen.

Dieser Algorithmus wird durch die Funktion **readany** veranschaulicht. Ihr Argument ist ein Vektor namens **fds**, der **nfds** Filedeskriptoren enthält. Die Funktion kehrt solange nicht zurück, bis wenigstens ein **read**-Aufruf auf einen dieser Filedeskriptoren ein Zeichen liefert. Dann liefert sie den Index des Filedeskriptors in **fds** zurück, von dem das Zeichen gelesen wurde. Das Zeichen selbst ist der Funktionswert. Der Aufrufer von **readany** sammelt vermutlich die hereinkommenden Daten auf irgendeine sinnvolle Weise zur späteren Verarbeitung. Um unparteiisch zu sein, merken wir uns, wo wir aufgehört haben, so daß wir die nächste Abfragerunde von dort aus starten.[2]

```
int readany(fds, nfds, whichp)  /* nach Eingabe fragen */
    int fds[], nfds, * whichp;
{
    static int lasti;
    int i;
    char c;

    i = lasti + 1;
    while (1)
    {
        if (i >= nfds)
            i = 0;
        setblock(fds[i], FALSE);
        switch (read(fds[i], &c, 1))
        {
            case -1:
                syserr("read");
            case 0:
                if (i == lasti)
                    sleep(1);
                i++;
                continue;
```

[2]Nehmen Sie an, daß die Laborgeräte Zeilen hereinschicken, die mit *Newlines* beendet sind, so dumm dies auch erscheinen mag. In Abschnitt 4.4.8 werden wir sehen, wie man Daten liest, ohne darauf warten zu müssen, daß eine ganze Zeile fertig ist.

```
            default:
                * whichp = lasti = i;
                return(c & 0377);  /* Vorzeichenuebertrag verhindern */
        }
    }
}
```

Der Aufruf von **sleep**, um die Zeit verstreichen zu lassen, besitzt zwei Nachteile: Erstens könnte eine Verzögerung von ungefähr einer Sekunde entstehen, bevor ein hereinkommendes Zeichen verarbeitet wird, was bei zeitkritischen Anwendungen zu Problemen führen kann. Zweitens könnte es sein, daß wir aufwachen und eine weitere vergebliche Runde drehen, und dies vielleicht viele Male, bis ein Zeichen vorliegt. Am besten wäre ein Systemaufruf der sagt: 'Lege mich solange schlafen, bis auf irgendeinem Filedeskriptor ein Eingabezeichen vorliegt.' Leider bietet UNIX keine solche Einrichtung, obwohl ein kundenspezifischer Gerätetreiber dies sehr wohl könnte.

Da diese Funktion mehrere Ereignisse nahezu gleichzeitig bearbeitet (datenübertragende Terminals), stellt sie eine primitive Form von Multitasking dar. Obwohl dieses Beispiel durchaus sinnvoll ist, wollen wir nicht übertreiben, indem wir Do-it-yourself-Multitasking lehren — schließlich ist UNIX bereits selbst ein Multitasking-System. Im allgemeinen ist es einfacher, solche Ereignisse durch mehrere Prozesse abzuarbeiten, als einen einzelnen Multitasking-Prozeß zu codieren, der Techniken wie blockierungsfreie Eingabe verwendet. Im Falle unserer Laborgeräte hätten wir zum Beispiel einen Prozeß programmieren können, der gerade ein einziges Gerät bearbeitet und hätten dann den UNIX-Kern die Prozesse für uns simultan ablaufen lassen können. Tatsächlich arbeitet UNIX gewöhnlich genau so, wobei die 'Geräte' die sich die Rechenzeit teilenden Benutzer darstellen.

Manchmal wollen wir sogar nur wissen, ob ein Zeichen vorliegt, ohne es zu lesen. Später dann, wenn wir bereit sind, wollen wir das Zeichen übernehmen. Unglücklicherweise kann **read** dies nicht tun; wenn ein Zeichen anliegt, so wird es auch geliefert. Wir können unser Vorhaben durchführen, indem wir das voreilig gelesene Zeichen in einem Puffer ablegen. Dann könnte unsere Funktion, die wir aufrufen um ein Zeichen zu holen, zuerst im Puffer nachschauen. Wenn das Zeichen vorhanden ist, liefern wir es; falls nicht, schalten wir die Eingabe auf Blockierung um und rufen **read** auf. Dieses Schema wird in den zwei nachfolgenden Funktionen verwendet: **cready** sagt uns einfach, ob ein Zeichen anliegt, und **cget** holt ein Zeichen; falls nötig, wird auch darauf gewartet.

```
#define EMPTY '\0'
static char cbuf = EMPTY;

BOOLEAN cready()            /* ist ein Zeichen abholbereit? */
{
    if (cbuf != EMPTY)
        return(TRUE);
    setblock(0, FALSE);
    switch (read(0, &cbuf, 1))
    {
        case -1:
            syserr("read");
        case 0:
            return(FALSE);  /* koennte auch EOF sein */
        default:
            return(TRUE);
    }
}

int cget()                  /* ein Zeichen lesen */
{
    char c;

    if (cbuf != EMPTY)
    {
        c = cbuf;
        cbuf = EMPTY;
        return(c & 0377);   /* Vorzeichenuebertrag verhindern */
    }
    setblock(0, TRUE);
    switch (read(0, &c, 1))
    {
        case -1:
            syserr("read");
        case 0:
            return(-1);     /* muss EOF sein */
        default:
            return(c & 0377);
    }
}
```

Wir haben das Nullbyte mit der Bedeutung eines leeren Puffers vorbelegt (**EMPTY**), was bedeutet, daß von **cready** gelesene Nullbytes ignoriert werden. Falls dies nicht zufriedenstellt, so kann man eine separate **BOOLEAN**-Variable als Anzeige für einen leeren Puffer verwenden.

Beachten Sie, wie **cready** und **cget** zwischen blockierungsfreier und blockierender Eingabe hin- und herschalten. Da wir nur aus einer einzigen Eingabequelle lesen, besteht keine Notwendigkeit, wie bei **readany** im blockierungsfreien Zustand zu bleiben, während wir prüfen, ob ein Zeichen verfügbar ist. Wir schalten einfach wieder auf blockierende Eingabe um und rufen **read** auf. Schließlich ist das Warten auf ein Zeichen genau das, was blockierende Eingabe bedeutet.

Am nützlichsten sind **cready** und **cget** dann, wenn ein Programm gewisse Arbeiten eben irgendwann erledigen muß, diese aber warten können, wenn ein Zeichen vorliegt. Ein gutes Beispiel dafür ist ein Programm, das **curses** (das Bildschirmausgabe-Paket von Berkeley) benützt.[3] Zur Veränderung des Schirminhalts wird eine Anzahl von Unterprogrammen aufgerufen, die einen Puffer mit der Größe des Bildschirms modifizieren. Danach, wann immer es geht, wird **refresh** aufgerufen, um den physikalischen Bildschirm zu verändern. Dies ist zeitraubend, da **refresh** den augenblicklichen Schirminhalt mit dem neuen Aussehen vergleicht, um die Zahl der zu übertragenden Zeichen zu minimieren. Ein flinker Tipper kann den Bildschirmmodifikationen leicht vorauseilen, besonders dann, wenn der Schirm bei jedem Tastendruck verändert werden muß, was bei einem Bildschirmeditor der Fall ist. Eine elegante Lösung besteht darin, **refresh** aus der Eingaberoutine aufzurufen, doch nur dann, wenn die Eingabe sich nicht im Wartezustand befindet. Muß sie dagegen warten, so hat der Benutzer offensichtlich vorausgetippt, ohne daß der Bildschirm nachgekommen ist, so daß die Auffrischung des Bildschirm ausgelassen wird. Der Schirminhalt wird dann modifiziert, wenn die Eingaberoutine das nächste Mal ausgeführt wird, falls dann nicht wieder ein Zeichen wartet. Wenn der Benutzer aufhört zu tippen, wird der Schirminhalt angepaßt. Wenn andererseits das Programm Zeichen schneller verarbeiten kann als der Tipper sie eingibt, dann wird der Schirm bei jedem Tastendruck modifiziert.

Dies mag kompliziert klingen, doch indem wir Funktionen benutzen, die wir bereits entwickelt haben, gibt dies nur drei Zeilen Programmtext:

```
if (!cready())
    refresh();
c = cget();
```

[3] **curses** ist Bestandteil von 4.1 BSD; doch viele Händler und 4.2 BSD erwerben es in Lizenz von Berkeley und packen es in ihre Nicht-Berkeley-Versionen (z.B. ist es bei Xenix 3 und PC/IX enthalten). Es ist beinahe immer verfügbar, wenn der Bildschirmeditor **vi** heißt.

4.4 Der Systemaufruf "ioctl"

```
#include <termio.h>

int ioctl(fd, cmd, tbuf)    /* Steuerung des Terminalgeraets */
    int fd;                 /* Filedeskriptor                */
    int cmd;                /* Kommando                      */
    struct termio * tbuf;   /* Terminalinformation           */

/* liefert 0 bei Erfolg oder -1 bei Fehler                   */

int ioctl(fd, cmd, arg)
    int fd;
    int cmd;
    int arg;                /* Kommandoargument              */

/* liefert 0 bei Erfolg oder -1 bei Fehler                   */
```

Für Terminals gibt es zwei Formen des Aufrufs von **ioctl**.[4] In der ersten Form ist das dritte Argument ein Zeiger auf eine Struktur, die Information über das Terminal enthält. Diese Struktur wird entweder als Eingabeparameter übergeben oder durch den Funktionsaufruf gefüllt. In der zweiten Form ist das dritte Argument nur eine Integer-Flagge. Andere Nicht-Terminal-Gerätetreiber mögen das dritte Argument anders definieren — dies bleibt dem Entwickler des Gerätetreibers überlassen.

4.4.1 Grundlegende Anwendungen von ioctl

Die Struktur, die die Terminalinformationen enthält (**struct termio**), beinhaltet ungefähr 50 Bits, die dem Treiber erzählen, wie hereinkommende und herausgehende Zeichen zu verarbeiten sind, wie Parameter von Datenübertragungsleitungen (wie z.B. die Baud-Rate) zu setzen sind usw. Außerdem legt die Struktur einige Steuerzeichen fest, wie *Erase* (normalerweise #), *Kill* (normalerweise @) und *EOT* (normalerweise *Control*-D). Nun die Struktur aus **/usr/include/termio.h**:

```
#define NCC 8
struct termio
{
    unsigned short  c_iflag;    /* Eingabeflaggen            */
    unsigned short  c_oflag;    /* Ausgabeflaggen            */
    unsigned short  c_cflag;    /* Kontrollflaggen           */
    unsigned short  c_lflag;    /* Line-discipline-Flaggen   */
```

[4] Der Name steht im Widerspruch zu **fcntl** — kürzt man '*control*' mit 'ctl' oder 'cntl' ab?

```
    char            c_line;     /* Line-discipline          */
    unsigned char   c_cc[NCC];  /* Steuerzeichen            */
};
```

Bevor wir in die Tiefen dieser Struktur einsinken, wollen wir die zur Verfügung stehenden Kommandos untersuchen:

TCGETA Füllen der Struktur **termio** mit Informationen über das Terminal, auf das **fd** verweist.

TCSETA Terminal aufgrund der in der Struktur enthaltenen Information einstellen.

TCSETAW Ähnlich dem vorigen Kommando, doch wird solange gewartet, bis alle noch nicht verarbeiteten Zeichen gesendet sind. Dies sollte gemacht werden, wenn Änderungen vorgenommen werden, die die Ausgabe betreffen. So wird sichergestellt, daß zuvor ausgegebene Zeichen unter den Bedingungen verarbeitet werden, unter denen sie geschrieben wurden.

TCSETAF Ähnlich dem vorigen Kommando, doch wird die Eingabe-Warteschlange entleert (Zeichen werden weggeworfen). Wird ein neuer interaktiver Zustand angenommen, wie es z.B. beim Starten eines Bildschirmeditors der Fall ist, so ist dies das sicherste Kommando, da es eine unbeabsichtigte Wirkung von eventuell vorausgetippten Zeichen verhindert.

Die nächsten drei Kommandos verwenden die zweite Form von **ioctl**:

TCSBRK Warten, bis die Ausgabe-Warteschlange leer ist. Wenn **arg** 0 ist, wird *break* (eine Viertelsekunde lang 0-Bits) geschickt.

TCXONC Wenn **arg** 0 ist, wird die Ausgabe sofort unterbrochen. Ist **arg** 1, so wird sie wiederaufgenommen.

TCFLSH Hat **arg** den Wert 0, so wird die Eingabe-Warteschlange entleert, bei 1 die Ausgabe-Warteschlange und bei 2 beide.

Eine beachtlich detaillierte Beschreibung der Struktur **termio** finden Sie im UNIX-Handbuch im Abschnitt über Gerätetreiber, üblicherweise unter 'tty' oder 'termio'. Wir wollen hier nicht alle Einzelheiten wiederholen, sondern nur die wichtigeren unter ihnen. Bestimmte Flaggen werden wir nur über ihre symbolischen Namen ansprechen. Befragen Sie Ihr Handbuch nach den wirklichen Bitmustern.

4.4.2 Geschwindigkeit, Zeichengröße und Parität

Flaggen innerhalb von **c_cflag** setzen die Baud-Rate (von 50 bis 9600), die Zeichengröße (5 bis 8 Bits), die Zahl der Stop-Bits (1 oder 2) und die Parität (gerade, ungerade oder keine). Dies ist kein Feld für schöpferische Betätigung — man ist normalerweise völlig damit zufrieden, wenn man eine Einstellung findet, die auch funktioniert.

Zusätzlich kann man Informationen über Zeichen erhalten, die durch die Paritätsprüfung gefallen sind. Ist die **PARMRK**-Flagge gesetzt, so werden den fehlerhaften Zeichen die Zeichen 0377 und 0 vorangestellt. Anderenfalls werden Zeichen mit Paritätsfehler als Nullbytes eingelesen. Alternativ dazu kann die **IGNPAR**-Flagge gesetzt werden, um Zeichen mit Paritätsfehlern gänzlich zu ignorieren.

4.4.3 Zeichenabbildung

Die Abbildung von *Newline* auf *Return* und von *Return* auf *Newline* bei der Eingabe wird durch die Flaggen **INLCR** und **ICRNL** innerhalb von **c_iflag** kontrolliert. Bei Terminals, die bei Drücken der *Return*-Taste sowohl *Return* als auch *Newline* zum Rechner schicken, kann *Return* ignoriert werden (**IGNCR**).

Bei der Ausgabe verlangen wir üblicherweise, daß *Newline* in ein *Return-Newline*-Paar umgewandelt wird. Diese Arbeit wird von der Flagge **ONLCR** innerhalb von **c_oflag** erledigt. Andere Flaggen bewirken, daß *Return* in *Newline* umgewandelt und *Return* in Spalte 0 unterdrückt wird. Falls *Newline* auch *Return* bewirkt, so gibt es eine Flagge, die dies dem Terminaltreiber mitteilt, so daß dieser über die Spaltennummer auf dem laufenden bleibt (wegen *Tab* und *Backspace*).

Der Treiber kann auch Terminals behandeln, deren Zeichensatz nur Großbuchstaben umfaßt; diese sind heutzutage weitaus weniger verbreitet als früher. Falls **XCASE** innerhalb von **c_lflag**, **IUCLC** innerhalb von **c_iflag** und **OLCUC** innerhalb von **c_oflag** gesetzt sind, so werden Großbuchstaben bei der Eingabe in Kleinbuchstaben und Kleinbuchstaben bei der Ausgabe in Großbuchstaben umgewandelt. Dies ist wünschenswert, da UNIX Kleinbuchstaben häufiger benützt als Großbuchstaben. Zur Ein- oder Ausgabe von Großbuchstaben wird dem betreffenden Buchstaben ein *Backslash* (Rückwärtsschrägstrich: \) vorangestellt.

Ist die **ISTRIP**-Flagge innerhalb von **c_iflag** gesetzt, so werden Eingabezeichen (nachdem die Parität geprüft worden ist) auf sieben Bits verkürzt, sonst werden alle acht Bits eingelesen. Die meisten Terminals verwenden den ASCII-Zeichensatz, der ein sieben-Bit-Code ist, so daß die Verkürzung normalerweise erwünscht ist. Es kann jedoch auch spezialisierte Geräte geben, die alle acht Bits übertragen. Bei der Ausgabe werden alle vom Prozeß gesendeten Bits ausgegeben. Sollen acht Datenbits auf das Terminal geschickt werden, so muß die Paritätserzeugung durch Löschen der **PARENB**-Flagge innerhalb von **c_cflag** ausgeschaltet werden.

4.4.4 Verzögerungen und Tabulatorzeichen

Einige Terminals, insbesondere mechanische, benötigen zur Durchführung bestimmter Operationen, wie z.B. dem Wagenrücklauf, gewisse Zeit. Man kann innerhalb von **c_oflag** Flaggen setzen, um die Verzögerungen für *Newline*, *Return*, *Backspace*, horizontale und vertikale Tabulatoren und *Formfeed* anzupassen.

Eine weitere Flagge namens **TAB3** bewirkt, daß ausgegebene Tabulatorzeichen durch die passende Zahl von Leerzeichen ersetzt werden. Dies ist dann sinnvoll, wenn das Terminal selbst keine Tabulatoreinrichtung besitzt, wenn es zu mühsam ist, sie zu setzen, oder wenn das Terminal in Wirklichkeit ein anderer Computer ist, dem die Ausgabe geschickt wird, und wenn deshalb nur Leerzeichen erwünscht sind.

4.4.5 Flußsteuerung

Ein Benutzer kann die Ausgabe auf ein Terminal vorübergehend anhalten, indem er *Control*-S drückt; ein Prozeß erreicht dies durch Aufruf von **ioctl** mit dem **TCXONC**-Kommando (siehe Abschnitt 4.4.1). Die Ausgabe wird durch den Benutzer durch Drücken von *Control*-Q oder durch **ioctl** fortgesetzt.

Ist die **IXANY**-Flagge innerhalb von **c_iflag** gesetzt, so kann der Benutzer durch Drücken eines beliebigen Zeichens — nicht nur *Control*-Q — die Ausgabe fortsetzen. Ist die **IXON**-Flagge nicht gesetzt, so steht dem Benutzer überhaupt keine Ausgabeflußsteuerung zur Verfügung; *Control*-S und *Control*-Q haben dann keine spezielle Bedeutung.

Der Terminaltreiber unterstützt auch Eingabeflußsteuerung. Ist die **IXOFF**-Flagge gesetzt, so sendet der Treiber, falls die Eingabe-Warteschlange überläuft, *Control*-S an das Terminal, um die Eingabe zu unterbrechen. Nimmt die Länge der Warteschlange ab, weil ein Prozeß einige der angestauten Zeichen liest, so wird *Control*-Q geschickt, um dem Terminal mitzuteilen, daß es mit der Eingabe fortfahren soll. Diese Eigenschaft kann natürlich nur bei solchen Terminals genutzt werden, die dies unterstützen.

4.4.6 Steuerzeichen

Einige Steuerzeichen kann man ändern, indem man Elemente des Vektors **c_cc** innerhalb der Struktur **termio** setzt. Die folgende Tabelle zeigt für jedes einstellbare Steuerzeichen den Index in **c_cc** und den jeweiligen Vorgabewert:

Zeichen	*Index*	*Vorgabewert*
Interrupt	0	*DEL*
Quit	1	*Control-\ (FS)*
Erase	2	#
Kill	3	@
EOT	4	*Control-D (EOT)*
EOL	5	*NUL*

Auf vielen Systemen unterscheiden sich die Vorgabewerte für *Erase* und *Kill* von den von uns gezeigten Werten. Zum Beispiel könnte *Erase* gleich *Backspace* und *Kill* gleich *Control*-U sein. *EOL* ist ein Zeilenbegrenzer (wie *Newline*), wird jedoch nur selten benutzt.

Um ein Steuerzeichen zu unterdrücken, können Sie es auf oktal 377 setzen. Alternativ dazu können Sie *Interrupt* und *Quit* durch Löschen der **ISIG**-Flagge innerhalb von **c_lflag** unterdrücken.

4.4.7 Echo

Terminals laufen normalerweise im Duplex-Betrieb, was bedeutet, daß Daten gleichzeitig in beiden Richtungen über die Übertragungsleitung fließen können. Folglich erzeugt der Computer, nicht das Terminal, die Wiedergabe (das Echo) von Zeichen, damit eine Kontrollmöglichkeit zur Verfügung steht, ob die Zeichen korrekt empfangen wurden. Dabei finden die üblichen Zeichenabbildungen für die Ausgabe statt, so daß z.B. *Return* in *Newline* und dann bei der Wiedergabe *Newline* in *Return* und *Newline* umgewandelt wird.

Um das Echo abzuschalten, wird die **ECHO**-Flagge innerhalb von **c_lflag** gelöscht. Dies wird entweder gemacht um Geheimhaltung zu garantieren, wie z.B. beim Tippen eines Paßworts, oder weil der Prozeß selbst entscheiden muß, ob überhaupt ein Echo erfolgen soll und wenn ja, was wo wiedergegeben werden soll (wie z.B. bei der Ausführung eines Bildschirmeditors).

Für die Zeichen *Erase* und *Kill* stehen zwei spezielle Arten von Echo zur Verfügung. Wenn das *Erase*-Zeichen *Backspace* ist, so kann die **ECHOE**-Flagge gesetzt werden, um *Backspace* als *Backspace*-Leerzeichen-*Backspace* wiederzugeben. Dies hat den angenehmen Effekt, daß gelöschte Zeichen vom Bildschirm verschwinden (bei Schreibstationen hat es keine Wirkung, außer einem Wackeln des Typenträgers). Man kann die **ECHOK**-Flagge setzen, um nach einem *Kill*-Zeichen *Newline* (das möglicherweise auf *Return* und *Newline* abgebildet ist) auszugeben; dadurch erhält der Benutzer eine frische Zeile, auf der er weiterarbeiten kann.

4.4.8 Verzögerungsfreie Eingabe

Normalerweise werden Eingabezeichen gesammelt, bis eine Zeile vollständig ist, was durch *Newline* oder durch *EOT* signalisiert wird. Erst dann stehen **read** irgendwelche Zeichen zur Verfügung, welches womöglich nur ein Zeichen verlangt hat und dieses dann bekommt. In vielen Anwendungen, wie z.B. bei Bildschirmeditoren oder formatgebundenen Systemen, möchte der lesende Prozeß die Zeichen in dem Moment zur Verfügung haben, in dem sie getippt wurden — ohne darauf warten zu müssen, bis eine Zeile zusammengesetzt ist. Tatsächlich hat dann der Begriff einer 'Zeile' vielleicht überhaupt keine Bedeutung mehr.

Wenn die **ICANON**-Flagge ('*canonical*': kanonisch) gelöscht ist, so werden die Eingabezeichen nicht zu Zeilen zusammengesetzt, bevor sie gelesen werden, deshalb ist Editieren mittels *Erase* und *Kill* nicht möglich — beide Zeichen haben dann ihre spezielle Bedeutung verloren. Zwei Parameter, **MIN** und **TIME**, bestimmen, wann ein **read**-Aufruf zufriedengestellt ist. Hat die Warteschlange eine Länge von **MIN** Zeichen erreicht oder sind **TIME** Zehntel Sekunden verstrichen, so werden die Zeichen in der Warteschlange verfügbar.

Die Positionen von *EOT* und *EOL* im Vektor **c_cc** (Indizes 4 und 5) dienen zur Aufbewahrung von **MIN** und **TIME**, da diese Positionen bei gelöschter **ICANON**-Flagge nicht für ihre üblichen Zwecke benötigt werden. Vergewissern Sie sich, daß Sie **MIN** und **TIME** gesetzt haben, da Sie sonst die aktuellen *EOT*- und *EOL*-Zeichen für **MIN** und **TIME** erhalten, was zu sehr merkwürdigen Ergebnissen führen wird. Falls Sie jemals auf einen Prozeß stoßen, der nach jedem vierten getippten Zeichen die Eingabe erhält, so ist wahrscheinlich gerade obiges passiert. (*Control*-D, das übliche *EOT*-Zeichen, hat den Wert 4).

Die Idee bei **MIN** und **TIME** ist, einem Prozeß zu erlauben, Zeichen zu erhalten, sobald oder kurz nachdem sie getippt wurden, ohne dabei den Nutzen zu verlieren, mit einem einzigen **read**-Aufruf mehrere Zeichen lesen zu können. Solange Sie jedoch Ihre Eingaberoutine nicht so codieren, daß die Eingabe gepuffert wird, können Sie **MIN** genausogut auf 1 setzen (**TIME** ist dann bedeutungslos), da Ihre **read**-Aufrufe ohnehin nur ein einzelnes Zeichen lesen werden.

Nachfolgend zeigen wir Ihnen eine puffernde, doch verzögerungsfreie, Eingaberoutine namens **keystroke**. Da sie die Terminalflaggen verändert, haben wir die zugehörige Hilfsroutine **restore** zur Verfügung gestellt, die das Terminal wieder auf die Einstellung bringt, die es vor dem ersten Aufruf von **keystroke** besaß. Das aufrufende Programm muß **restore** aufrufen, bevor es endet.

```
#include <termio.h>

static struct termio tbufsave;

char keystroke()  /* Zeichen vom Terminal lesen */
{
    static char buf[10];
    static int total, next;
    static BOOLEAN first = TRUE;
    struct termio tbuf;

    if (first)
    {
        first = FALSE;
        if (ioctl(0, TCGETA, &tbuf) == -1)
            syserr("ioctl");
        tbufsave = tbuf;
        tbuf.c_lflag &= ~ICANON;
        tbuf.c_cc[4] = sizeof(buf);  /* MIN  */
        tbuf.c_cc[5] = 2;            /* TIME */
        if (ioctl(0, TCSETAF, &tbuf) == -1)
            syserr("ioctl2");
    }
    if (next >= total)
        switch (total = read(0, buf, sizeof(buf)))
        {
            case -1:
                syserr("read");
            case 0:
                fatal("Mysterious EOF");
            default:
                next = 0;
        }
    return(buf[next++]);
}

void restore()  /* Terminalflaggen restaurieren */
{
    if (ioctl(0, TCSETAF, &tbufsave) == -1)
        syserr("ioctl3");
}
```

Beachten Sie, daß wir nur **ICANON** gelöscht (und natürlich **MIN** und **TIME** gesetzt) haben. Typischerweise würde man mehr Flaggen löschen, das Echo ausschalten, Zeichenumwandlung bei der Ausgabe vornehmen usw. Dieser Aufgabe werden wir uns im nächsten Abschnitt widmen.

Für spezielle Anwendungen wollen Sie vielleicht mit den Werten für **MIN** und **TIME** ein wenig experimentieren. Wir benutzten 10 Zeichen und 0.2 Sekunden, was gut zu klappen scheint.

4.5 Nackte Ein- und Ausgabe für Terminals

In Version 7 und früheren Versionen konnte man eine einzelne Flagge löschen, um das Terminal in den sogenannten *raw*-Modus zu setzen, was bedeutet, daß keine besondere Ein- oder Ausgabeverarbeitung vorgenommen wird, und daß Zeichen sofort lesbar sind, ohne daß man darauf warten muß, bis eine Zeile zusammengebaut ist. Seit damals sind die Verhältnisse viel komplizierter geworden. Zwar besitzt das **stty**-*Kommando* immer noch eine **raw**-Option, doch um den gleichen Effekt mittels **ioctl** zu erreichen, müssen mehrere Flaggen angepaßt werden.

Wir hätten gerne, daß nackte Ein- und Ausgabe für Terminals folgende Merkmale besitzt:

1. *Verzögerungsfreie Eingabe.* Löschen Sie **ICANON** und setzen Sie **MIN** und **TIME**.
2. *Keine Zeichenumwandlung.* Löschen Sie **OPOST** innerhalb von **c_oflag**, um die Ausgabeverarbeitung abzuschalten. Für die Eingabe löschen Sie **INLCR**, **ICRNL** und **IUCLC**. Löschen Sie **ISTRIP**, um alle acht Bits zu bekommen.
3. *Keine Flußsteuerung.* Löschen Sie **IXON**.
4. *Keine Steuerzeichen.* Löschen Sie **BRKINT** und **ISIG**.
5. *Kein Echo.* Löschen Sie **ECHO**.

Diese Operationen sind in der Funktion **setraw** zusammengefaßt. Beachten Sie, daß sie die alte **termio**-Struktur zur Verwendung durch **restore** sichert.

```
void setraw()  /* Terminal in raw-Modus setzen */
{
    struct termio tbuf;

    if (ioctl(0, TCGETA, &tbuf) == -1)
        syserr("ioctl");
```

```
    tbufsave = tbuf;
    tbuf.c_iflag &= ~(INLCR | ICRNL | IUCLC | ISTRIP | IXON | BRKINT);
    tbuf.c_oflag &= ~OPOST;
    tbuf.c_lflag &= ~(ICANON | ISIG | ECHO);
    tbuf.c_cc[4] = 5;  /* MIN  */
    tbuf.c_cc[5] = 2;  /* TIME */
    if (ioctl(0, TCSETAF, &tbuf) == -1)
        syserr("ioctl2");
}
```

Da wir nun im Besitz von **setraw** sind, können wir **keystroke** so umschreiben, daß es **setraw** aufruft, anstatt **ICANON** selbst zu löschen. Üblicherweise verlangen Programme, die verzögerungsfreie Eingabe fordern, auch die anderen Eigenschaften nackter Ein- und Ausgabe.

Manchmal passiert es beim (oder nach dem!) Testen von Programmen, daß ein Programm, das das Terminal in den *raw*-Modus gesetzt hat, abbricht, bevor es die ursprünglichen Werte wiederhergestellt hat. Programmierer denken dann oft, daß der Computer abgestürzt ist oder daß sich das Terminal verklemmt hat — sie sind nicht einmal mehr in der Lage, *EOT* zu benützen, um sich abzumelden. Es ist jedoch möglich, aus dem *raw*-Modus herauszukommen. Rufen Sie sich zunächst in Erinnerung, daß **ICRNL** gelöscht ist; das bedeutet, daß Sie Ihre Eingabezeilen mit der *Linefeed-(Newline-)* Taste oder mit *Control*-J abschließen müssen, anstatt die *Return*-Taste zu benutzen.[5] Außerdem werden Sie das, was Sie tippen, nicht sehen, da auch **ECHO** ausgeschaltet ist. Beginnen Sie damit, einige *Linefeed*-Zeichen zu tippen — Sie sollten dann einige *Prompt*-Zeichen der Shell sehen, jedoch nicht am linken Rand, da die Ausgabeumwandlung (**OPOST**) ausgeschaltet ist. Dann tippen Sie folgendes, gefolgt von einem *Linefeed*-Zeichen:

```
stty echo icanon icrnl opost
```

Die Kommunikation mit dem Terminal sollte nun wesentlich verbessert sein, obwohl Ihnen immer noch *Erase*, *Kill*, Flußsteuerung und einige andere Dinge fehlen. Sie können sie entweder durch ein weiteres **stty**-Kommando einschalten oder sich einfach abmelden.

Einige UNIX-Systeme besitzen bei **stty** eine Option namens **sane**, die alle Flaggen zugleich auf vernünftige Werte einstellt, so daß Sie auf diesen Systemen die Flaggen sogar noch schneller wiederherstellen können.[6]

[5] Eine angenehme Verbesserung der Shell bestünde darin, daß sie sowohl *Return* als auch *Newline* als Zeilenende akzeptiert.

[6] Allerdings setzt **sane** die Parität auf *even*... A.d.Ü.

4.6 Andere Gerätedateien

Abgesehen von Gerätedateien für Platten und Terminals werden UNIX-Systeme normalerweise mit mehreren anderen Gerätedateien ausgeliefert — darunter solche für Floppylaufwerke, Magnetbandstationen, Zeilendrucker, Konsole und Hauptspeicher. Diese unterscheiden sich von System zu System, so daß Sie gezwungen sind, Ihr spezielles UNIX-Handbuch zu befragen, um herauszufinden, wie diese zu benutzen sind.

In Abschnitt 4 oder 7 des Handbuchs Ihres UNIX-Systems sollte ein kurzer Steckbrief jeder Gerätedatei vorhanden sein. Der Gebrauch von **open**, **close**, **read** und **write** ist im allgemeinen einfach. Die Schwierigkeit liegt bei **ioctl**, wie wir im Falle der Terminals gesehen haben. Der Entwickler eines Gerätetreibers kann diesen Systemaufruf verwenden, wie es ihm gerade einfällt, und diese Einfälle sind gelegentlich etwas sonderbar. Trotzdem besitzen Sie ein Recht darauf, daß Definitionsdateien dokumentiert (und ihrem System beigefügt) sind und daß alle relevanten **ioctl**-Kommandos vollständig dokumentiert sind.

4.7 Portabilität

Wir haben die Funktion von Terminals bei System III und System V beschrieben und bei Systemen, die auf diesen Versionen basieren, wie Xenix 3. Doch selbst bei diesen Systemen können Sie möglicherweise feststellen, daß Erweiterungen vorgenommen wurden. Wenn z.B. Ihr Computer eine integrierte Konsole besitzt, kann es sein, daß zusätzliche Optionen für **ioctl** zur Kontrolle der Tastatur und des Bildschirms mit aufgenommen wurden (z.B. damit jedesmal, wenn eine gewisse Zahl von Zeilen auf den Bildschirm ausgegeben wurde, eine Pause eingelegt wird).

Die anderen wichtigen UNIX-Versionen verhalten sich ähnlich, soweit es **open**, **close**, **read** und **write** betrifft (in Version 7 fehlt **O_NDELAY**). Die wichtigsten Abweichungen finden sich bei **ioctl** und der damit verbundenen C-Struktur.

In Version 7 benützt **ioctl** verschiedene Kommandos, und die C-Struktur ist in der Definitionsdatei **sgtty.h** vereinbart.[7] Die dort vereinbarte Struktur ist von **termio** völlig verschieden, doch die Prinzipien sind dieselben. Wenn Sie der Darlegung dieses Kapitels gefolgt sind, so werden Sie mit Version 7 wenig Schwierigkeiten haben, da deren Terminal-Gerätetreiber viel einfacher sind als die von System III und System V.

Wie zu erwarten war, ist Berkeley 4.2 BSD noch stärker abgedriftet (die Definitionsdatei heißt dort **ioctl.h**). Doch auch hier sind die Prinzipien vergleichbar und Sie sollten in der Lage sein, Probleme mittels des Steckbriefs zu **tty** im Abschnitt 4 des 4.2 BSD-Handbuchs zu lösen.

[7] Ältere Versionen besaßen Systemaufrufe namens **stty** und **gtty**. In Version 7 sind diese aus Kompatibilitätsgründen enthalten, doch seit System III wird auf sie verzichtet.

Falls Sie sich nicht auf System III, System V und die darauf basierenden Versionen beschränken, können Sie keine portablen Programme schreiben, die **ioctl** direkt verwenden. Statt dessen sollten Sie herausfinden, welche Eigenschaften Sie wirklich benötigen, wie z.B. das Setzen des Terminals in den *raw*-Modus, und Funktionen mit einer einheitlichen Schnittstelle für vielfältige Implementierungen codieren, wie wir es in Abschnitt 3.14 gezeigt haben. Es ist jammerschade, daß mit UNIX niemals Unterprogramme mitgeliefert wurden (ähnlich zu **setraw** und **restore**), mit denen man die üblichen Aufgaben portabel erledigen kann.

Eine weitere Möglichkeit ist, Techniken von der Funktion **mkdir2** aus Abschnitt 3.8 zu übernehmen und das **stty**-*Kommando* auszuführen, da dieses portabler ist als **ioctl**:

```
system("stty raw");
```

Obgleich dies bei **mkdir2** elegant erschien, scheint es hier bestenfalls ein böser Trick zu sein. Dies wahrscheinlich deswegen, weil wir in diesem Fall Alternativen haben. Bei **mkdir2** hatten wir keine andere Wahl — wir mußten einen Sohn-Prozeß erzeugen, um die effektive Benutzernummer auf die des Super-Users zu setzen.

Der vorgeschlagene Standard von */usr/group* ist bei Terminals überhaupt keine Hilfe. Anscheinend gab es zu viele Unterschiede, als daß der Ausschuß zu einer Einigung gelangen konnte. **ioctl** steht zwar in obigem Standard (als Platzhalter), doch die zugehörigen Argumente sind nicht erklärt. Anstatt sich auf die verschiedenen Bits zu einigen, wäre es praktischer, eine Menge 'höherwertigerer' Funktionen in der Art von **setraw** und **restore** zu definieren. Jeder Entwickler könnte diese dann passend implementieren, und die Programme könnten diese 'höherwertigeren' Funktionen verwenden anstatt **ioctl** selbst. Es ist zweifelhaft, ob */usr/group* diese Richtung einschlägt. In der Vergangenheit hat */usr/group* Eigenschaften von bestehenden UNIX-Implementierungen (in erster Linie System III) ausgewählt, anstatt neue Systemaufrufe oder Unterprogramme zu erfinden.

Unterhält sich Ihr Programm mit einem Bildschirmterminal, so sollten Sie unter allen Umständen versuchen, **curses** zu verwenden (das auch die Tastatur behandelt). Es ist nicht nur über die UNIX-Versionen, in denen es enthalten ist, sondern auch über mehr als 100 verschiedene Terminalarten hinweg ziemlich portabel.

Viele Personal-Computer besitzen schnelleren Zugriff auf den Bildschirm als die Behandlung dessen als Terminal, indem sie den von der Ausgabesteuerung verwendeten Bildschirmspeicher direkt auf den Adreßraum der CPU abbilden. Dies wird *Memory-mapped-Video* genannt. Auf einem einfachen Computer ist es einem UNIX-Prozeß möglich, auf diesen Speicher direkt zuzugreifen, doch auf Computern mit Speicherverwaltungseinheit ist ein solcher Zugriff verboten.[8] Der Hersteller mag

[8] Ich verwende den Bildschirmeditor EDIX in PC/IX. Um den Schirm zu modifizieren, schaltet EDIX einfach die Interrupts ab und mißbraucht die Segmentierungsregister, um auf den Bildschirmspeicher zuzugreifen, der ein ganzes Stück außerhalb seines Adreßraums liegt. Der IBM PC ist machtlos, ihn aufzuhalten.

stattdessen einen Gerätetreiber zur Verfügung stellen, der schnellen Zugriff auf den Bildschirm bietet — schnelleren Zugriff, als wenn die herkömmlichen *Escape*-Sequenzen verwendet werden. Eine solche Fähigkeit erlaubt die Programmierung einer ansprechenderen Benutzer-Schnittstelle, und macht Anwendungen wie Tabellenkalkulationen realisierbar. Doch dadurch werden die Portabilitätsprobleme sogar noch verschlimmert, da *Memory-mapped-Video* viel weniger standardisiert ist als *Escape*-Sequenzen. Und selbst wenn der Computer *Memory-mapped-Video* besitzt, so kann es nur dazu verwendet werden, den eingebauten Bildschirm anzusprechen, nicht aber die Bildschirme von entfernt aufgestellten Terminals.

In den meisten Fällen ist es wahrscheinlich am besten, alle Bildschirme wie Terminals zu behandeln, selbst wenn ein Bildschirm über *Memory-mapped-Video* betrieben werden kann. Unglücklicherweise benachteiligt dies UNIX-Systeme im Vergleich zu einfacheren Personal-Computern (selbstverständlich überwogen von anderen Faktoren). Eine gute Lösung für das Problem des Bildschirmzugriffs zu finden, ist eine der entscheidensten Entwicklungsanforderungen, denen die UNIX-Gemeinde gegenübersteht. Die Antwort umfaßt wahrscheinlich drei Bereiche: eine Neudefinition der Fähigkeiten, die ein Terminal besitzen soll (z.B. seitliches Verschieben des Bildschirminhalts), (wieder einmal!) eine Neufassung des UNIX-Terminaltreibers und Überlegungen, wie Speicherverwaltung auf einem Computer mit integriertem Bildschirm funktionieren sollte. (Die in Abschnitt 7.7 beschriebenen Systemaufrufe für Shared-Memory könnten hierbei von Nutzen sein).

Aufgaben

4.1 Ändern Sie **getln2**, wie in Abschnitt 4.2 besprochen, um Zeilen zurückzuliefern, die mit *EOT* abgeschlossen sind.

4.2 Implementieren Sie **Sfdopen**, wie in Abschnitt 4.2 besprochen.

4.3 Implementieren Sie eine vereinfachte Version des **stty**-Kommandos, das einfach den aktuellen Terminalstatus ausgibt.

4.4 Implementieren Sie eine Version des **stty**-Kommandos, die es einem Benutzer erlaubt, nur die 10 am häufigsten benutzten Flaggen zu ändern. Sie müssen entscheiden, welche 10 das sind.

4.5 Implementieren Sie einen einfachen Bildschirmeditor. Entscheiden Sie sich für eine kleine Auswahl an Funktionen, die vernünftiges Editieren gestatten, aber ohne allzuviel Drumherum. Halten Sie den zu editierenden Text im Hauptspeicher. Suchen Sie sich ein Terminal, auf das Sie Zugriff haben, und modifizieren Sie den Bildschirm durch Verwendung geeigneter *Escape*-Sequenzen.

4.6 Falls Sie Zugriff darauf haben, benützen Sie das **curses**-Paket (siehe Abschnitt 4.3) zur Lösung von Aufgabe 4.5.

Kapitel 5: Prozesse

5.1 Einführung

Wir verlassen nun das Gebiet der Ein- und Ausgabe, um die Multitasking-Möglichkeiten zu erforschen, die UNIX bietet. Dieses Kapitel beschäftigt sich mit Techniken zum Aufruf von Programmen und Prozessen unter Verwendung von **exec**, **fork**, **wait** und verwandten Systemaufrufen. Im nächsten Kapitel wird einfache Prozeßkommunikation mit Hilfe von Pipes erklärt, und Kapitel 7 fährt mit weiter fortgeschrittenen Mechanismen der Prozeßkommunikation fort.

Unsere Darstellung folgt dem Ziel der Implementierung eines ziemlich vollständigen Kommandointerpreters (auch *Shell* genannt). Wir beginnen mit einer Shell mit begrenzten Fähigkeiten, die nur in sehr eingeschränktem Maße benutzbar ist. Dann fügen wir nach und nach neue Merkmale hinzu, bis wir schließlich im nächsten Kapitel eine Shell besitzen, die Ein- und Ausgabeumlenkung, Pipelines, Hintergrundprozesse, mit Anführungszeichen versehene Argumente und Environment-Variablen beherrscht.

5.2 Das Environment

Zu Beginn wollen wir das *Environment* behandeln, mit dem die meisten UNIX-Benutzer bereits auf der Shell-Ebene vertraut sind. Da das UNIX-Handbuch jedoch versäumt, zu dokumentieren, wie man das Environment ändert, besteht unsere erste Aufgabe darin, diese Lücke zu schließen.

Wird ein UNIX-Programm ausgeführt, so erhält es vom aufrufenden Prozeß zweierlei Datenmengen: die *Argumente* und das Environment. Bei C-Programmen handelt es sich jeweils um einen Vektor mit Zeigern auf Zeichen, dessen letztes Element ein Nullzeiger ist. Alle anderen Elemente verweisen auf eine durch ein Nullbyte abgeschlossene Zeichenkette. Weiterhin wird die Zahl der Argumente als Zähler übergeben. Andere Sprachen unter UNIX, wie Fortran oder Pascal, benützen andere Schnittstellen, doch wir sind hier nur an C interessiert.

Ein C-Programm beginnt wie folgt:

```
extern char ** environ;

main(argc, argv, envp)
    int argc;
    char * argv[], * envp[];
```

Sowohl **envp** als auch **environ** enthalten den gleichen Wert, so daß **envp** normalerweise weggelassen wird. Jede Zeichenkette des Arguments kann alles mögliche sein, solange sie nur mit einem Nullbyte abgeschlossen ist. Zeichenketten des Environments unterliegen stärkeren Beschränkungen. Per Konvention besitzt jede Zeichenkette die Form *variable=value*, mit einem Nullbyte nach der Wertangabe. Environment-Variablen werden normalerweise durch ein Zuweisungskommando der Shell gesetzt, etwa wie folgt:

```
$ PATH=/bin:/usr/bin:/usr/marc/bin::
```

Ein Prozeß kann den Wert einer Environment-Variablen mit Hilfe der Standardfunktion **getenv** bestimmen, die wir bereits früher im Abschnitt 3.4 benutzten, um den Wert der Variablen **TZ** zu erhalten. Alternativ kann ein Prozeß durch den Zeiger **environ** direkt auf das Environment zugreifen. Dies wird durch das folgende einfache Programm **envprint** verdeutlicht, das das gesamte Environment ausgibt:

```
extern char ** environ;

main(argc, argv)
    int argc;
    char * argv[];
{
    int i;

    for (i = 0; environ[i] != NULL; i++)
        printf("%s\n", environ[i]);
    exit(0);
}
```

Hier ist die Ausgabe, die wir erhielten:

```
HOME=/usr/marc
LOGNAME=marc
MAIL=/usr/mail/marc
PATH=/bin:/usr/bin:/usr/marc/bin::
```

```
PS1=$
PS2=>
TZ=MST7MDT
```

Das *Verändern* des Environments ist dagegen nicht so einfach. Obwohl der durch das Environment belegte interne Speicherplatz ausschließliches Eigentum des Prozesses ist und deshalb beliebig modifiziert werden kann, gibt es keinen zusätzlichen Raum für irgendwelche neuen Variablen oder für Werte, die mehr Platz benötigen. Deshalb muß, einmal abgesehen von primitiven Veränderungen, ein völlig neues Environment geschaffen werden. Läßt man dann den Zeiger **environ** auf das neue Environment zeigen, so wird dieser an jedes im Anschluß aufgerufene Programm übergeben, wie wir in Abschnitt 5.3 noch sehen werden.

Muß eine große Menge an Veränderungen vorgenommen werden, so ist es am bequemsten, das ganze Environment aus der komprimierten Darstellung in eine Datenstruktur zu kopieren, die flexibel genug ist, daß neue Variablen hinzugefügt und größere Werte zugewiesen werden können. Eine solche Datenstruktur wird *Symboltabelle* genannt. Bevor dann ein neues Programm aufgerufen wird, können wir eine Version in der komprimierten Darstellung erzeugen und **environ** geeignet setzen. Dies ist auch gerade das, was **sh** (die Standard-Shell) macht. Sie benötigt diese Flexibilität zur Verarbeitung von Zuweisungskommandos.

Da wir in diesem und im nächsten Kapitel unsere eigene Shell implementieren möchten, benötigen wir ebenfalls eine Symboltabelle für das Environment. Wir betrachten ein Beispiel für eine Symboltabellen-Struktur und die dazugehörigen Zugriffsfunktionen. Zum Zugriff auf die Variablen verwenden wir ein lineares Suchverfahren, was bei kleinen Symboltabellen durchaus angemessen ist. In anderen Situationen kann man zur Beschleunigung der Suche Hash-Verfahren oder binäre Bäume verwenden, was jedoch zu Lasten der Komplexität und der Codegröße geht.[1]

Da **sh** sowohl eine Programmiersprache als auch ein Kommandointerpreter ist, werden in der Symboltabelle viele Variablen gespeichert, die nur für dasjenige Shell-Programm von Interesse sind, das gerade ausgeführt wird (z.B. Schleifenzählvariablen oder Variablen, die Dateinamen enthalten). Die Übergabe dieser *lokalen* Variablen an andere Programme wäre unrentabel und möglicherweise irreführend. Zur Unterscheidung zwischen *lokalen* und *globalen* Variablen, die in das Environment eingetragen werden sollen, erlaubt **sh**, daß eine Variable mit **export** gekennzeichnet wird. Dazu wird etwa folgendes Kommando verwendet:

```
export PATH
```

[1] Weitergehendere Informationen über Symboltabellen-Techniken finden Sie in jedem guten Lehrbuch über Datenstrukturen oder Compilerbau, wie z.B. in *Fundamentals of Data Structures* von Ellis Horowitz und Sartaj Sahni (Potomac, Md. : Computer Science Press, Inc. , 1976).

Wird ein Environment aus der Symboltabelle aufgebaut, so enthält es das ursprüngliche Environment und dazu noch die neuen Werte exportierter Variablen. Veränderungen von nicht exportierten Variablen werden ignoriert.

Das Exportieren funktioniert nur in eine Richtung: von einem Prozeß zu irgendeinem Programm, das dieser aufruft. Da jedes Programm eine Kopie des Environments erbt (in seinem eigenen Adreßraum), ist es einem Prozeß unmöglich, über das Environment Daten an seinen Vater zurückzureichen.

Als Symboltabelle wollen wir folgenden Vektor verwenden:

```
#define MAXVAR 25

static struct varslot
{
    char * name;        /* Variablenname                    */
    char * val;         /* Variablenwert                    */
    BOOLEAN exported;   /* soll Variable exportiert werden? */
} sym[MAXVAR];
```

Dieser Vektor ähnelt dem Vektor **environ** mit der Ausnahme, daß wir aus Gründen vereinfachten Zugriffs und besserer Änderbarkeit die Zeichenketten für den Variablennamen und den zugehörigen Wert getrennt haben. Außerdem haben wir eine Flagge namens **exported** eingebaut, die uns angibt, welche Variablen wir berücksichtigen müssen, wenn wir das Environment in komprimierter Form aufbauen. Der Vektor bietet Platz für 25 Variablen, was für unsere Beispiel-Programme ausreicht, für eine richtige Shell jedoch zuwenig sein mag. Ein freier Platz wird dadurch gekennzeichnet, daß der Zeiger **name** den Wert **NULL** besitzt. Da externe Variablen auf Null initialisiert sind, sind zu Beginn alle Plätze frei.

Zur Zuweisung eines Werts an eine Variable durchsuchen wir einfach den Vektor **sym** und halten Ausschau nach einer Struktur **varslot** mit einer passenden **name**-Komponente. Haben wir sie gefunden, so ändern wir die entsprechende Komponente **val**. Falls nicht, so nehmen wir den ersten freien Platz und speichern dort den Namen und den zugehörigen Wert. Gibt es keine freien Plätze mehr, so ist die Symboltabelle voll. Die Aufgabe, einen passenden oder freien Platz zu finden, wird durch die Funktion **find** vorgenommen.

```
static struct varslot * find(name)  /* Symboltabellen-Eintrag */
    char * name;                    /* finden                 */
{
    int i;
    struct varslot * v;

    v = NULL;
```

```
    for (i = 0; i < MAXVAR; i++)
        if (sym[i].name == NULL)
        {
            if (v == NULL)
                v = &sym[i];
        }
        else if (strcmp(sym[i].name, name) == 0)
        {
            v = &sym[i];
            break;
        }
    return(v);
}
```

find liefert einen Zeiger auf einen Platz oder **NULL**, wenn die Symboltabelle voll ist. Das Prüfen der Komponente **name**, auf die gezeigt wird, muß der Aufrufer übernehmen; ist diese **NULL**, so wurde kein passender Name gefunden und es steht dem Aufrufer frei, eine neue Variable und den zugehörigen Wert auf diesem Platz zu speichern.

Das Äquivalent zum Zuweisungskommando von **sh** wird von der Funktion **EVset** wahrgenommen, deren Argumente zwei Zeichenketten sind: Der Name einer Variablen und der ihr zuzuweisende Wert. **EVset** ruft **find** auf, um den (vorhandenen oder freien) Platz zu bestimmen, und benützt dann **assign**, um die eigentliche Arbeit des Kopierens von Name und Wert vorzunehmen.

```
BOOLEAN EVset(name, val)  /* Namen und Wert zum      */
    char * name, * val;   /* Environment hinzufuegen */
{
    struct varslot * v;

    if ((v = find(name)) == NULL)
        return(FALSE);
    return(assign(&v->name, name) && assign(&v->val, val));
}
```

Erinnern Sie sich daran, daß bei **find** der Resultatwert **NULL** bedeutet, daß der Name noch nicht in der Symboltabelle enthalten ist und es keinen freien Platz gibt, um ihn aufzunehmen. Deshalb mißlingt **EVset** in diesem Fall.

Als nächstes betrachten wir die Funktion **assign**. Ihr erstes Argument ist ein Zeiger auf eine **name**-Komponente bzw. **val**-Komponente, das zweite Argument ist die Zeichenkette, die den neuen Namen bzw. Wert enthält. Ist die Komponente gegenwärtig **NULL**, so legt **malloc** genügend Platz zur Aufbewahrung der Zeichenkette an. Anderenfalls existiert bereits Platz zur Aufbewahrung des alten Werts

und mit Hilfe von **realloc** wird sichergestellt, daß er groß genug ist. Danach wird die Zeichenkette in die angelegte Speicherfläche kopiert. Da **realloc** eventuell die Speicherfläche verlagern muß, kann es passieren, daß der an **assign** übergebene Zeiger (**name** oder **val**) verändert wird. Aus diesem Grund wird an **assign** ein Zeiger auf die Komponente übergeben; dies ist die übliche Technik in C zur Rückgabe eines Werts durch ein Argument.

Hier ist der Code für **assign**:

```
char * malloc(), * realloc();

static BOOLEAN assign(p, s)  /* Name oder Wert initialisieren */
    char ** p, * s;
{
    int size;

    size = strlen(s) + 1;
    if (* p == NULL)
    {
        if ((* p = malloc(size)) == NULL)
            return(FALSE);
    }
    else if ((* p = realloc(* p, size)) == NULL)
        return(FALSE);
    strcpy(* p, s);
    return(TRUE);
}
```

Zum Setzen der **exported**-Flagge rufen wir **EVexport** auf. Die einzige Schwierigkeit dabei besteht darin, daß die Variable, die wir zum Exportieren anlegen, möglicherweise noch nicht existiert. In diesem Fall weisen wir ihr eine leere Zeichenkette als Wert zu, um einen Platz für sie zu reservieren.

```
BOOLEAN EVexport(name)  /* zu exportierende Variable setzen */
    char * name;
{
    struct varslot * v;

    if ((v = find(name)) == NULL)
        return(FALSE);
    if (v->name == NULL)
        if (!assign(&v->name, name) || !assign(&v->val, ""))
            return(FALSE);
```

```
    v->exported = TRUE;
    return(TRUE);
}
```

Das Beschaffen eines Werts aus der Symboltabelle ist einfach: Die Funktion **EVget** gleicht der Funktion **getenv**, ausgenommen, daß sie auf die Symboltabelle zugreift anstatt auf das komprimierte Environment, auf das **environ** zeigt:

```
char * EVget(name)  /* Variablenwert bestimmen */
    char * name;
{
    struct varslot * v;

    if ((v = find(name)) == NULL || v->name == NULL)
        return(NULL);
    return(v->val);
}
```

Normalerweise würde ein Programm die Symboltabelle aus dem geerbten Environment initialisieren. Da das Programm per Definition nur exportierte Variablen erhielt, sollten alle geerbten Variablen in der Symboltabelle entsprechend markiert werden.[2] Diese Initialisierung wird von der Funktion **EVinit** vorgenommen, die vor allen anderen Symboltabellen-Funktionen aufgerufen werden sollte.

```
BOOLEAN EVinit()  /* Symboltabelle aus          */
{                 /* Environment initialisieren */
    int i, namelen;
    char name[20];

    for (i = 0; environ[i] != NULL; i++)
    {
        namelen = strcspn(environ[i], "=");
        strncpy(name, environ[i], namelen);
        name[namelen] = '\0';
        if (!EVset(name, &environ[i][namelen + 1]) || !EVexport(name))
            return(FALSE);
    }
    return(TRUE);
}
```

[2] Diese Behandlung exportierter Variablen unterscheidet sich ein wenig von der Behandlung durch sh.

Die Standardfunktion **strcspn** liefert hier die Zahl der Zeichen, die dem ersten = in einer Zeichenkette des Environments vorausgehen.

Nun entnehmen wir der Symboltabelle alle exportierten Variablen und bauen ein neues Environment in komprimierter Form, das an andere Programme übergeben werden kann. Dabei verwenden wir den Speicherplatz des geerbten Environments nicht, da er möglicherweise nicht groß genug ist (es könnte ja sein, daß wir zusätzliche Variablen dazugefügt oder einige Werte vergrößert haben). Aus diesem Grund legen wir beim erstmaligen Aufbau eines Environments Speicherplatz für einen Vektor von Zeigern und für die Zeichenketten an, auf die verwiesen wird. Damit wir nicht alles neu anlegen müssen, wenn wir das nächste Mal ein Environment aufbauen, setzen wir die **updated**-Flagge. Beim nächsten Mal müssen wir nur **realloc** verwenden, um sicherzustellen, daß der Platz für die Zeichenketten groß genug ist. Der Vektor selbst wird in Ordnung sein. Der Aufbau wird durch die Funktion **EVupdate** vorgenommen:

```
BOOLEAN EVupdate()  /* Environment aus Symboltabelle bauen */
{
    int i, envi, nvlen;
    struct varslot * v;
    static BOOLEAN updated = FALSE;

    if (!updated)
        if ((environ = (char **)malloc((MAXVAR + 1)
             * sizeof(char *))) == NULL)
            return(FALSE);
    envi = 0;
    for (i = 0; i < MAXVAR; i++)
    {
        v = &sym[i];
        if (v->name == NULL || !v->exported)
            continue;
        nvlen = strlen(v->name) + strlen(v->val) + 2;
        if (!updated)
        {
            if ((environ[envi] = malloc(nvlen)) == NULL)
                return(FALSE);
        }
        else
            if ((environ[envi] = realloc(environ[envi], nvlen)) == NULL)
                return(FALSE);
```

```
            sprintf(environ[envi], "%s=%s", v->name, v->val);
            envi++;
        }
        environ[envi] = NULL;
        updated = TRUE;
        return(TRUE);
    }
```

Da wir die Symboltabelle auch ausgeben möchten, fügen wir schließlich noch eine weitere Funktion hinzu:

```
    void EVprint()  /* Environment ausgeben */
    {
        int i;

        for (i = 0; i < MAXVAR; i++)
            if (sym[i].name != NULL)
                printf("%3s %s=%s\n", sym[i].exported ? "[E]" : "",
                    sym[i].name, sym[i].val);
    }
```

Den ausgegebenen Variablen wird ein in eckigen Klammern gesetztes **E** vorangestellt, falls sie exportiert werden. Diese Methode unterscheidet sich von der der Shell, welche ein (argumentloses) Kommando namens **set** zur Ausgabe aller Variablen und ihrer Werte sowie ein (argumentloses) Kommando namens **export** zur Ausgabe der Namen aller exportierten Variablen zur Verfügung stellt.

Um Ihnen die Verwendung dieser Symboltabellen-Funktionen zu veranschaulichen, zeigen wir Ihnen ein kleines Programm, das zwei Variablen in die Symboltabelle einträgt und die Symboltabelle ausgibt:

```
    main()
    {
        if (!EVinit())
            fatal("can't initialize environment");
        printf("Before update:\n");
        EVprint();
        if (!EVset("count", "0"))
            fatal("EVset");
        if (!EVset("BOOK", "/usr/marc/book") || !EVexport("BOOK"))
            fatal("EVset or EVexport");
        printf("\nAfter update:\n");
        EVprint();
        exit(0);
    }
```

Hierzu die Ausgabe:

```
Before update:
[E] TZ=MST7MDT
[E] PS2=>
[E] PS1=$
[E] PATH=/bin:/usr/bin:/usr/marc/bin::
[E] MAIL=/usr/mail/marc
[E] LOGNAME=marc
[E] HOME=/usr/marc

After update:
[E] TZ=MST7MDT
[E] PS2=>
[E] PS1=$
[E] PATH=/bin:/usr/bin:/usr/marc/bin::
[E] MAIL=/usr/mail/marc
[E] LOGNAME=marc
[E] HOME=/usr/marc
    count=0
[E] BOOK=/usr/marc/book
```

In unserem Beispiel haben wir weder **EVupdate** noch irgendein Programm aufgerufen, an das wir unser verändertes Environment vererben konnten. Dies werden wir im nächsten Abschnitt vornehmen, sobald wir gelernt haben, wie man ein Programm aufruft.

5.3 Die "exec"-Systemaufrufe

Es ist unmöglich, die Systemaufrufe **exec** und **fork** zu verstehen, wenn man den Unterschied zwischen einem *Prozeß* und einem *Programm* nicht völlig verstanden hat. Sind Ihnen diese Begriffe neu, so sollten Sie vielleicht zurückblättern und den Abschnitt 1.3 kurz wiederholen. Wenn Sie dagegen gleich weitermachen wollen, fassen wir den Unterschied in einem Satz zusammen: Ein *Prozeß* stellt eine Ausführungsumgebung dar und besteht aus Anweisungssegment, Benutzer-Datensegment und System-Datensegment, während ein *Programm* eine Datei ist, die Anweisungen und Daten zur Initialisierung des Anweisungssegments und des Benutzer-Datensegments eines Prozesses enthält.

Die **exec**-Systemaufrufe dienen zur erneuten Initialisierung eines Prozesses durch ein bestimmtes Programm; das Programm ändert sich, während der Prozeß derselbe bleibt. Demgegenüber erzeugt der Systemaufruf **fork** einen neuen Prozeß als

Abbild eines existenten Prozesses, indem einfach Anweisungssegment, Benutzer-Datensegment und System-Datensegment kopiert werden. Der neue Prozeß wird also nicht von einem Programm initialisiert.[3] **fork** wird Gegenstand von Abschnitt 5.4 sein.

Ohne **fork** ist die Verwendung von **exec** nur von begrenztem Nutzen; ohne **exec** gibt **fork** überhaupt keinen Sinn. Denken Sie daran, wenn wir Ihnen die beiden Systemaufrufe getrennt vorstellen, und seien Sie nicht beunruhigt, wenn sie Ihnen sinnlos erscheinen — versuchen Sie einfach nur zu verstehen, was sie tun. Wenn wir sie in Abschnitt 5.4 gemeinsam verwenden, werden Sie sehen, daß sie ein leistungsfähiges Paar bilden. Eine gute Frage ist, weshalb man zwei Systemaufrufe, die nur bei gemeinsamer Verwendung sinnvoll sind, nicht in einen einzigen Systemaufruf integriert hat. Wir werden Ihnen eine ausgezeichnete Antwort darauf geben, doch fehlt uns zum Verständnis dieser Antwort momentan noch der nötige Hintergrund, den wir Ihnen in Kapitel 6 liefern werden.

Abgesehen vom Starten des UNIX-Kerns selbst, stellt **exec** die einzige Möglichkeit dar, unter UNIX Programme auszuführen. Nicht nur die Shell benützt **exec** zur Ausführung unserer Programme, auch die Shell und ihre Vorfahren wurden durch **exec** aufgerufen. Und **fork** schließlich stellt die einzige Möglichkeit dar, neue Prozesse zu erzeugen.

Es gibt sechs **exec**-Systemaufrufe, doch beschreiben wir zunächst nur einen von ihnen ausführlich. Anschließend machen wir Sie mit den anderen vertraut und erklären, worin sie sich unterscheiden.

```
                    /* Programm ausfuehren                    */
int execl(path, arg0, arg1, ..., argn, null)
    char * path;  /* Pfad der Programmdatei                   */
    char * arg0;  /* erstes Argument (Dateiname)              */
    char * arg1;  /* zweites Argument                         */
    ...
    char * argn;  /* letztes Argument                         */
    char * null;  /* NULL                                     */

/* kehrt zurueck bei Fehler (Resultatwert ist dann -1) */
```

Das Argument **path** muß eine Programmdatei benennen, die für die effektive Benutzernummer ausführbar ist (d.h. z.B. die Rechte oktal 755 besitzt) und den Anforderungen eines ausführbaren Programms genügt. Das Anweisungssegment und das Benutzer-Datensegment des Prozesses, der **execl** aufruft, werden durch die Anweisungen und die Daten des Programms überlagert. Anschließend führt der Prozeß

[3] Die AT&T Handbücher von System III und System V verwirren den Leser durch die Behauptung, daß exec "den aufrufenden Prozeß in einen neuen Prozeß verwandelt." Dies ist falsch — es gibt keinen neuen Prozeß.

das neue Programm ab dessen Anfangsposition aus (d.h. die Funktion **main** wird aufgerufen).

Von einem erfolgreichen **execl**-Aufruf gibt es keine Rückkehr, da der Punkt für den Rücksprung verschwunden ist. Ein mißlungener **execl**-Aufruf kehrt dagegen mit einem Wert von –1 zurück; es besteht jedoch keine Notwendigkeit, diesen Wert zu prüfen, da ja kein anderer Wert möglich ist. Die häufigsten Ursachen für einen mißlungenen **execl**-Aufruf sind darin begründet, daß **path** nicht existiert oder die Datei nicht ausführbar ist, was meistens daran liegt, daß der Benutzer einen Kommandonamen falsch getippt hat.

Die weiteren Argumente von **execl** werden in einen Vektor von Zeigern auf Zeichen zusammengefaßt; das letzte Argument, das **NULL** sein muß, beendet den Vorgang. Per Konvention ist das erste dieser Argumente der Name der Programmdatei (nicht der ganze Pfadname). Das neue Programm kann auf diese Argumente durch die bekannten Argumente **argc** und **argv** von **main** zugreifen. Auch das Environment, auf das **environ** verweist, wird mit übergeben und ist durch den Zeiger **environ** des neuen Programms erreichbar.

Da der Prozeß weiterlebt und sein System-Datensegment zum größten Teil unversehrt geblieben ist, sind die allermeisten Prozeßattribute unverändert, darunter auch die Prozeßnummer, die Vater-Prozeßnummer, die Prozeßgruppennummer, die wirkliche Benutzernummer und Gruppennummer, der Arbeitskatalog und der Wurzelkatalog, die Priorität, die angesammelten Ausführungszeiten und üblicherweise auch die aktiven Filedeskriptoren. Es ist einfacher, die Attribute aufzuführen, die sich, alle aus gutem Grund, verändern:[4]

- Wurde der Prozeß angewiesen, Signale abzufangen, so werden diese auf die Standardreaktion eingestellt, da die Anweisungen zum Abfangen nicht mehr vorhanden sind. Ignorierte Signale oder Signale, die auf die Standardreaktion eingestellt waren, bleiben in diesem Zustand. (Mehr über Signale erfahren Sie in Kapitel 8.)

- Ist das *set-user-ID*-Bit oder das *set-group-ID*-Bit der neuen Programmdatei gesetzt, so wird die effektive Benutzernummer bzw. die effektive Gruppennummer auf die Benutzernummer bzw. Gruppennummer der Datei gesetzt.[5] Es gibt keine Möglichkeit, die früheren effektiven Nummern zurückzuerhalten, falls diese sich von den wirklichen Nummern unterschieden.

- Wird bei einem Prozeß gerade ein Ausführungszeitprofil erstellt (siehe Abschnitt 9.8), so wird es abgeschaltet. Da Zeitprofile immer nur bei einem bestimmten Programm Sinn machen, würde sonst nur Unsinn dabei herauskommen.

[4] Siehe dazu auch Anhang A.

[5] Herzlichen Glückwunsch! Sie befinden sich erst in Kapitel 5 und verstehen bereits das Zitat am Anfang des Vorworts!

Um Ihnen die Verwendung von **execl** zu verdeutlichen, zeigen wir Ihnen hier ein ziemlich gestelltes Beispiel:

```
execltest()
{
    printf("The quick brown fox jumped over ");
    execl("/bin/echo", "echo", "the", "lazy", "dogs.", NULL);
    syserr("execl");
}
```

Beim ersten Versuch arbeitete die Funktion ganz ausgezeichnet und lieferte folgende Ausgabe:

```
The quick brown fox jumped over the lazy dogs.
```

Doch dann lenkten wir die Standard-Ausgabe in eine Datei, um sie in den Text dieses Kapitels einzufügen, und alles was wir dann bekamen war dies:

```
the lazy dogs.
```

Was ging schief? Nun, es stellt sich heraus, daß **stdio**, die Standard-Bücherei der Ein- und Ausgabe, zu der auch **printf** gehört, ihre Ausgabe in Blöcken zu 512 Bytes puffert (so wie wir es im Abschnitt 2.10 gemacht haben), wenn in eine Datei oder auf eine Pipe, nicht aber wenn auf das Terminal geschrieben wird.[6] Normalerweise bemerken wir dabei keine Probleme, da der letzte Puffer automatisch entleert wird, wenn der Prozeß endet. Doch der Prozeß war noch nicht beendet, als **execl** aufgerufen wurde, und der im Benutzer-Datensegment befindliche Puffer wurde überlagert, bevor er entleert werden konnte.

Das Problem kann entweder dadurch gelöst werden, daß man ungepufferte Ausgabe erzwingt, etwa so:

```
setbuf(stdout, NULL);
```

oder indem man den Puffer entleert, unmittelbar bevor man **execl** aufruft:

```
fflush(stdout);
```

Wie bereits erwähnt, bleiben aktive Filedeskriptoren normalerweise über einen **execl**-Aufruf hinweg erhalten. Wollen wir, daß ein Filedeskriptor nicht aktiv bleibt, so schließen wir ihn einfach vorher mittels **close**. Doch manchmal können wir dies nicht. Nehmen wir an, daß wir ein Programm aufrufen, das verlangt, daß alle Filedeskriptoren geschlossen sind (eine äußerst ungewöhnliche Forderung). Wir könnten dann folgendes versuchen:

[6] Der Effekt, daß die Ausgabe nur bruchstückhaft erscheint, ist auch bei Ausgabe auf das Terminal möglich, falls Zeilenpufferung stattfindet. A.d.Ü.

```
for (i = 0; i < 20; i++)
    close(i);  /* Fehler ignorieren */
execl(path, arg0, arg1, arg2, NULL);
syserr("execl");
```

Beachten Sie, daß wir hier alle Filedeskriptoren unbedingt schließen wollen, so daß es uns sogar egal ist, ob sie überhaupt aktiv waren — wir schließen sie auf jeden Fall. Dies ist einer der seltenen Fälle, in denen es angemessen ist, Fehlerresultate von Systemaufrufen zu ignorieren.

Alles ist wunderbar falls **execl** gelingt. Falls der Aufruf schiefgeht, bekommen wir die Fehlermeldung niemals zu sehen, da der Filedeskriptor 2 geschlossen wurde. Jetzt kann die *close-on-exec*-Flagge benützt werden, auf die wir in Abschnitt 3.13 gestoßen sind. Falls diese bei **fcntl** gesetzt ist, wird der Filedeskriptor bei erfolgreichem **execl** geschlossen, anderenfalls in Ruhe gelassen. Wir können diese Flagge entweder nur für wichtige Filedeskriptoren oder einfach für alle setzen, etwa wie folgt:

```
for (i = 0; i < 20; i++)
    fcntl(i, F_SETFD, 1);  /* Fehler ignorieren */
execl(path, arg0, arg1, arg2, NULL);
syserr("execl");
```

Die *close-on-exec*-Flagge wird praktisch niemals benutzt, da üblicherweise die drei ersten Filedeskriptoren über einen **exec**-Aufruf hinweg aktiv bleiben und die anderen 17 geschlossen werden sollen. Bei manchen Anwendungen, die aus mehreren Prozessen bestehen, kann es jedoch nötig sein, von dieser Konvention abzuweichen.

Die anderen fünf **exec**-Systemaufrufe bieten drei Merkmale, die bei **execl** nicht zur Verfügung stehen:

- Die Argumente werden in einen Vektor eingestellt und nicht explizit aufgeführt. Dies ist notwendig, wenn die Zahl der Argumente zum Übersetzungszeitpunkt unbekannt ist, wie es zum Beispiel bei der Programmierung eines Kommandoprozessors der Fall ist.

- Mit Hilfe des Werts der Variablen **PATH** wird nach der Programmdatei gesucht — so, wie es **sh** auch macht.

- Anstatt **environ** automatisch zu benützen, wird explizit ein Zeiger auf ein Environment übergeben. Da jedoch ein erfolgreicher **exec**-Aufruf das bestehende Environment ohnehin überschreibt, bietet diese Möglichkeit keinen besonderen Vorteil.[7]

[7] Es sei denn, exec geht schief. A.d.Ü.

Die Verteilung dieser Merkmale auf die sechs **exec**-Varianten spiegelt sich in der folgenden Tabelle wider:

System-aufruf	*Format des Argumente*	*Übergabe des Environments*	*Suche via PATH?*
execl	Liste	automatisch	nein
execv	Vektor	automatisch	nein
execle	Liste	von Hand	nein
execve	Vektor	von Hand	nein
execlp	Liste	automatisch	ja
execvp	Vektor	automatisch	ja

Da man beinahe immer automatische Environment-Übergabe und Suche mittels **PATH** haben will, stellt sich üblicherweise die Wahl zwischen **execlp** und **execvp**.

Im folgenden sehen Sie die Funktionsköpfe der **exec**-Varianten, die wir bisher noch nicht gezeigt haben:

```
int execv(path, argv)
    char * path;
    char * argv[];  /* Zeiger auf Argumente              */

int execle(path, arg0, arg1, ..., argn, null, envp)
    char * path;
    char * arg0;
    char * arg1;
    ...
    char * argn;
    char * null;
    char * envp[];  /* Zeiger auf Environment-Variablen */

int execve(path, argv, envp)
    char * path;
    char * argv[];
    char * envp[];
```

```
int execlp(file, arg0, arg1, ..., argn, null)
    char * file;  /* Name der Programmdatei */
    char * arg0;
    char * arg1;
    ...
    char * argn;
    char * null;

int execvp(file, argv)
    char * file;
    char * argv[];
```

Beachten Sie, daß das hier verwendete **argv**-Argument vom Aussehen her identisch ist mit dem **argv**-Argument einer **main**-Funktion. Ein kleiner Unterschied besteht darin, daß bei der Benutzung der **execv**-Varianten der letzte Zeiger **NULL** lauten muß, während C-Programmierern beigebracht wird, sich nach der Argumentzahl (**argc**) zu richten.[8]

Enthält ein **file**-Argument von **execlp** oder **execvp** keine Schrägstriche (/), so werden die im Wert der **PATH**-Variablen aufgeführten Zeichenketten nacheinander vor dieses Argument geklebt, bis eine gewöhnliche Datei mit dem resultierenden Pfadnamen gefunden wird, die ausgeführt werden darf. Wenn diese Datei ein Programm enthält (wie durch eine Codenummer in ihrem ersten Wort angezeigt wird), so wird sie ausgeführt. Falls nicht, so wird angenommen, daß sie eine Shell-Kommandodatei ist und **/bin/sh** mit dem Pfad als erstem Argument wird dazu benützt, sie auszuführen.[9]

Ergibt die Suche via **PATH** keine ausführbare Datei, so mißlingen **execlp** und **execvp**. Falls das **file**-Argument Schrägstriche beinhaltet, so wird nicht gesucht — es wird davon ausgegangen, daß der Pfad vollständig ist. Er kann jedoch immer noch eine Kommandodatei darstellen.

Es ist möglich, **execvp** und **execlp** als Unterprogramme zu programmieren, die **execv** oder **execl** aufrufen. Hier ist eine Version von **execvp**:

---

[8] In *The C Programming Language* ist nirgends aufgeführt, daß **argv[argc]** den Wert **NULL** darstellt. Ebenso verhält es sich bei den meisten UNIX-Handbüchern. Deshalb sollte man nicht annehmen, daß **argv** durch **NULL** abgeschlossen ist, da dies bei Nicht-UNIX-Systemen oder UNIX-Derivaten nicht der Fall zu sein braucht.

[9] Erstaunlicherweise ist in keinem der UNIX-Handbücher (egal welcher Version) in der Bibliothek des Autors erwähnt, daß **execlp** und **execvp** in der Lage sind, Kommandodateien zu verarbeiten, obwohl diese Tatsache unter UNIX-Systemprogrammierern ziemlich gut bekannt ist. Tatsächlich ist dieses Merkmal durch die C-Schnittstelle zu diesen Systemaufrufen implementiert, nicht durch den Kern selbst.

```
#define MAXPATH 50
#define MAXARGS 20

int execvp2(file, argv)
    char * file, * argv[];
{
    char * colon, * pathseq, path[MAXPATH], * newargv[MAXARGS];
    char * getenv(), * strchr();
    int i, len;
    extern int errno;

    if (strchr(file, '/') != NULL || (pathseq = getenv("PATH")) == NULL)
        pathseq = ":";
    for (; (colon = strchr(pathseq, ':')) != NULL; pathseq = colon + 1)
    {
        len = colon - pathseq;
        strncpy(path, pathseq, len);
        path[len] = '\0';
        if (len > 0)
            strcat(path, "/");
        strcat(path, file);
        execv(path, argv);
        if (errno == ENOEXEC)
        {
            i = 0;
            do
            {
                if (i >= MAXARGS - 1)
                {
                    errno = E2BIG;
                    return(-1);
                }
                newargv[i + 1] = argv[i];
            }
            while (argv[i++] != NULL);
            newargv[0] = "sh";
            execv("/bin/sh", newargv);
            syserr("execv");
        }
    }
    return(-1);
}
```

Die Standardfunktion **strchr** liefert einen Zeiger auf den Punkt, an dem ein Zeichen in einer Zeichenkette das erste Mal vorkommt, oder **NULL**, falls kein derartiges Zeichen enthalten ist. Wir benutzen sie zur schnellen Suche nach einem Schrägstrich und zum Auffinden des ersten Doppelpunkts.[10] Anstatt einen Dateinamen mit einem Schrägstrich als Spezialfall zu behandeln, verwenden wir einfach eine leere Pfadfolge. Dann entfernen wir nacheinander die Pfad-Vorsatzzeichen, bauen den vollen Pfad und versuchen dann diese Datei auszuführen. Haben wir damit keinen Erfolg, weil die Datei nicht in Form eines Programms vorlag, so nehmen wir an, daß sie eine Shell-Kommandodatei ist; in diesem Fall müssen wir die Argumente in einen neuen Vektor kopieren, um Platz für den neuen Kommandonamen (**sh**) zu schaffen. Gelang die Aktion aus irgendeinem anderen Grund nicht, so versuchen wir es mit einem anderen Pfad. Wir geben erst dann auf, wenn keine weiteren Pfad-Anwärter mehr übrig sind.

Wir kennen nun genügend Systemaufrufe, um eine Shell zu schreiben, wenn auch keine besonders gute. Das wesentliche Handicap einer solchen Shell besteht darin, daß sie zur Ausführung eines Kommandos Selbstmord begehen muß, da keine Rückkehr von **exec** möglich ist. Es ist zwar sinnlos, aber Sie können die Shell durch Eingabe falscher Kommandos am Leben erhalten. Da wir schon mal dabei sind, werden wir einige Kommandos zur Veränderung und zum Zugriff auf das Environment in die Shell einbauen: die Zuweisung (z.B. `BOOK=/usr/marc/book`), **set**, zur Ausgabe des *Environments*, und **export**, das die als Argumente gegebenen Variablen exportiert. Erhält **export** keine Argumente, so verhält es sich gerade wie **set**.

Unsere erste Aufgabe besteht darin, eine Kommandozeile in Argumente aufzuspalten. Der Einfachheit halber werden wir mit Anführungszeichen versehene Argumente nicht berücksichtigen. Darüberhinaus brauchen wir uns um die Sonderzeichen `&`, `;` und `|` nicht zu kümmern, da unsere Shell Hintergrundprozesse, Hintereinanderausführung und den Pipe-Mechanismus nicht beherrscht. Außerdem lassen wir die Umlenkung der Ein- und Ausgabe (`>`, `>>` und `<`) weg. Folglich besteht eine Kommandozeile einfach aus einer Reihe von Worten, die durch Leerzeichen oder Tabulatorzeichen voneinander getrennt sind. Wir brauchen sie nur aufzusammeln und in einen **argv**-Vektor zu stecken:

```
static BOOLEAN getargs(argcp, argv, max)  /* Argumente lesen und */
    int * argcp;                          /* trennen             */
    char * argv[];
    int max;
{
    static char cmd[100];
    char * cmdp, * strtok();
    int i;
```

[10] Damit dies richtig funktioniert, muß **PATH** am Ende Doppelpunkte besitzen. A.d.Ü.

```
    if (gets(cmd) == NULL)
        exit(0);
    cmdp = cmd;
    for (i = 0; i <= max; i++)
    {
        if ((argv[i] = strtok(cmdp, " \t")) == NULL)
            break;
        cmdp = NULL;  /* strtok soll weitergehen */
    }
    if (i > max)
    {
        printf("Too many args\n");
        return(FALSE);
    }
    * argcp = i;
    return(TRUE);
}
```

Da diese Funktion aus der Standard-Eingabe liest, haben wir sie so konstruiert, daß sie beim Lesen des Dateiendes einfach terminiert; dabei müssen vor der Termination keine Aufräumarbeiten erledigt werden. Die Standardfunktion **strtok** holt sich das nächste Symbol aus einer Zeichenkette. Beim ersten Aufruf wird ein Zeiger auf die Zeichenkette zur Verfügung gestellt; nachfolgende Aufrufe werden durch **NULL** als erstes Argument unterschieden. Das zweite Argument gibt die Folge der Zeichen an, die Symbole begrenzen — in diesem Falle Leerzeichen und Tabulatorzeichen. Zusätzlich zum Auffüllen des Vektors **argv** liefern wir durch den Zeiger **argcp** einen Argumentzähler, um dem Aufrufer die Mühe des Auszählens zu ersparen.

Als nächstes benötigen wir Funktionen zur Bearbeitung der drei eingebauten Kommandos Zuweisung, **set** und **export**. Diese sind einfach, da wir die Funktionen zur Manipulation des *Environments* aus dem vorigen Abschnitt verwenden können.

```
void asg(argc, argv)  /* Zuweisungskommando */
    int argc;
    char * argv[];
{
    char * name, * val, * strtok();

    if (argc != 1)
        printf("Extra args\n");
    else
```

```
    {
        name = strtok(argv[0], "=");
        val = strtok(NULL, "\1");  /* alles uebrige nehmen */
        if (!EVset(name, val))
            printf("Can't assign\n");
    }
}

void export(argc, argv)  /* export-Kommando */
    int argc;
    char * argv[];
{
    int i;

    if (argc == 1)
    {
        set(argc, argv);
        return;
    }
    for (i = 1; i < argc; i++)
        if (!EVexport(argv[i]))
        {
            printf("Can't export %s\n", argv[i]);
            return;
        }
}

void set(argc, argv)  /* set-Kommando */
    int argc;
    char * argv[];
{
    if (argc != 1)
        printf("Extra args\n");
    else
        EVprint();
}
```

Nun vervollständigen wir das Programm durch eine **main**-Funktion, die den Prompt ausgibt (wir verwenden den sogenannten "*prompt string* 2", der in der Environment-Variablen **PS2** aufbewahrt wird), die die Argumente holt, die prüft, ob das Kommando ein eingebautes Kommando ist und die versucht, es auszuführen, falls dies nicht zutrifft.

```
#define MAXARG 20

main()  /* Einweg-Shell */
{
    char * argv[MAXARG + 1], * prompt, * EVget();
    int argc;

    if (!EVinit())
        fatal("can't initialize environment");
    if ((prompt = EVget("PS2")) == NULL)
        prompt = "> ";
    while (1)
    {
        printf("%s", prompt);
        if (!getargs(&argc, argv, MAXARG) || argc == 0)
            continue;
        if (strchr(argv[0], '=') != NULL)
            asg(argc, argv);
        else if (strcmp(argv[0], "export") == 0)
            export(argc, argv);
        else if (strcmp(argv[0], "set") == 0)
            set(argc, argv);
        else
            execute(argc, argv);
    }
}

static void execute(argc, argv)  /* execute-Kommando */
    int argc;
    char * argv[];
{
    if (!EVupdate())
        fatal("can't update environment");
    execvp(argv[0], argv);
    printf("Can't execute\n");
}
```

Beachten Sie, daß wir die Schleife nur dann wiederholen und einen weiteren Prompt ausgeben, wenn **execvp** mißlingt. Aufgrund dieser Tatsache werden wir diese Shell vermutlich nicht sehr nützlich finden. Trotzdem zeigen wir Ihnen hier ein Beispiel einer Sitzung:

```
$ sh0
> BOOK=/usr/marc/book
> export BOOK
> envprint
HOME=/usr/marc
LOGNAME=marc
MAIL=/usr/mail/marc
PATH=/bin:/usr/bin:/usr/marc/bin::
PS1=$
PS2=>
TZ=MST7MDT
BOOK=/usr/marc/book
$
```

Das Kommando **envprint** war das Programm zur Ausgabe des Environments aus dem vorigen Abschnitt.

Wie bereits gesagt, wird **exec** normalerweise mit **fork** zusammen verwendet, doch manchmal ist es auch für sich alleine genommen nützlich. Manchmal wird ein großes Programm in mehrere Phasen aufgeteilt, und **exec** schaltet in eine neue Phase um. Da jedoch *alle* Anweisungen und Benutzerdaten überlagert werden, müssen die einzelnen Phasen völlig unabhängig voneinander sein, damit dies durchführbar ist — Daten können nur durch Argumente, über das Environment oder über Dateien übermittelt werden. Deshalb ist diese Anwendung selten, aber nicht unbekannt. Häufiger wird **exec** allein benutzt, wenn wir eine sehr kleine Menge an vorbereitenden Arbeiten vornehmen müssen, bevor ein Kommando ausgeführt wird. Ein Beispiel dafür ist das **nohup**-Kommando, das einfach *Hangup*-Signale ignoriert, bevor es **execvp** aufruft, um das Kommando des Benutzers auszuführen.[11] Ein weiteres Beispiel ist das Kommando **nice**, das die Priorität eines Kommandos verändert.

5.4 Der Systemaufruf "fork"

```
int fork()                          /* neuen Prozess erzeugen */

/* liefert Prozessnummer und 0 bei Erfolg oder -1 bei Fehler */
```

fork ist ein wenig das Gegenteil von **exec**: **fork** erzeugt einen neuen Prozeß, initialisiert ihn jedoch nicht durch ein neues Programm. Statt dessen sind Anweisungssegment, Benutzer-Datensegment und System-Datensegment des neuen Prozesses beinahe exakte Kopien der entsprechenden Segmente des alten Prozesses. Nach der

[11] Nein — **nohup** ist normalerweise eine Shell-Datei! A.d.Ü.

Rückkehr von **fork** erhalten beide Prozesse (Vater und Sohn) den Resultatwert. Die Resultatwerte sind jedoch verschieden, was sehr wichtig ist, da dadurch die nachfolgenden Aktionen unterschiedlich sein können. Üblicherweise führt der Sohn **exec** aus, und der Vater wartet entweder darauf, daß der Sohn terminiert, oder er macht irgendetwas anderes.

Der Sohn erhält von **fork** den Resultatwert 0; der Vater erhält die Prozeßnummer des Sohns. Wie üblich zeigt der Resultatwert -1 einen Fehler an, doch da **fork** keine Argumente besitzt, kann der Aufrufer nichts falsch gemacht haben. Die einzige mögliche Ursache für einen Fehler ist, daß gewisse Betriebsmittel aufgebraucht sind, z.B. daß nicht genug Platz zum Auslagern von Prozessen vorhanden ist, oder daß vom Aufrufer bereits zuviele Prozesse ausgeführt werden. Anstatt aufzugeben wird der Vater möglicherweise eine Weile warten (mittels **sleep**) und es nochmals versuchen, obwohl dies die Standard-Shell nicht macht — sie gibt einfach eine Meldung aus und meldet sich erneut mit einem Prompt.

Erinnern Sie sich daran, daß ein durch einen **exec**-Systemaufruf gestartetes Programm viele Attribute behalten hat, da das System-Datensegment weitgehend in Ruhe gelassen wurde. In ähnlicher Manier erbt ein durch **fork** erzeugter Sohn die meisten Attribute seines Vaters, da sein System-Datensegment von dem seines Vaters kopiert wird. Diese Hinterlassenschaft ermöglicht es einem Benutzer, gewisse Attribute (wie den Arbeitskatalog, die effektive Benutzernummer oder die Priorität) in der Shell zu setzen, welche dann für alle danach aufgerufenen Kommandos gelten. Man kann sich diese Attribute als zur "Sitzung" gehörend vorstellen, obwohl UNIX keine solche Vorstellung besitzt — die Hinterlassenschaft erübrigt dies.

Nur einige wenige Attribute werden nicht vererbt:[12]

- Offensichtlich sind die Prozeßnummer und die Vater-Prozeßnummer des Sohns verschieden, da der Sohn ein anderer Prozeß ist.
- Der Sohn erhält Kopien der aktiven Filedeskriptoren des Vaters. Beide sind mit der gleichen Datei verbunden und der zugehörige Positionszeiger besitzt den gleichen Wert. Der Positionszeiger wird gemeinsam benutzt. Ändert ihn der Sohn mittels **lseek**, so beginnt die nächste Lese- oder Schreiboperation des Vaters bei der neuen Position. Der Filedeskriptor selbst ist jedoch verschieden: Wenn ihn der Sohn schließt, bleibt die Kopie des Vaters davon unberührt.
- Die angesammelten Ausführungszeiten des Sohns sind auf Null zurückgesetzt, da dieser ja noch am Beginn seines Lebens steht.

[12] Siehe auch Anhang A.

Hier ist ein kleines Beispiel, das die Wirkung eines **fork**-Aufrufs zeigt:

```
forktest()
{
   int pid;

   printf("Start of test\n");
   pid = fork();
   printf("Returned %d\n", pid);
}
```

Die Ausgabe lautet:

```
Start of test
Returned 0
Returned 93
```

Wir können **fork** und **execvp** gemeinsam benutzen, um unsere Shell aus dem vorigen Abschnitt wesentlich brauchbarer zu machen — nachdem sie ein Kommando ausgeführt hat, wird sie nicht mehr einfach terminieren, sondern nun tatsächlich ein neues Kommando anfordern! Wir ersetzen die Funktion **execute** durch **execut2**:

```
static void execut2(argc, argv)  /* Kommando ausfuehren und warten */
   int argc;
   char * argv[];
{
   int code;

   switch (fork())
   {
       case -1:
          printf("Can't create new process\n");
          return;
       case 0:
          if (!EVupdate())
              fatal("can't update environment");
          execvp(argv[0], argv);
          fatal("can't execute");
       default:
          if (wait(NULL) == -1)
              syserr("wait");
   }
}
```

Die genauen Einzelheiten des Systemaufrufs **wait** werden wir ausführlich in Abschnitt 5.6 darstellen — für den Moment genügt es zu wissen, daß **wait** vor seiner Rückkehr auf das Ableben des Sohns wartet.

In dieser Funktion wird ein Fehler bei **fork** anders behandelt als ein Fehler bei **execvp**. Wenn **fork** –1 zurückliefert, haben wir keinen neuen Prozeß erzeugt, so daß wir nur eine Fehlermeldung ausgeben und zurückkehren. Der Aufrufer (die Funktion **main** aus dem vorigen Abschnitt) wird dann den Benutzer zur erneuten Kommandoeingabe auffordern — genau dies möchten wir ja. Doch wenn **execvp** zurückkehrt, so befinden wir uns beim *Sohn*-Prozeß und nicht beim Vater, so daß wir unbedingt den Prozeß terminieren. Andernfalls würde der Sohn weiterlaufen und wir hätten dann *zwei* Shell-Prozesse, die zur Kommandoeingabe auffordern. Wenn wir tatsächlich versuchen würden, ein Kommando einzugeben, so würden die beiden Prozesse um die Zeichen konkurrieren und jeder von ihnen würde ein paar von ihnen erhalten, da ein einzelnes Zeichen nur einmal gelesen werden kann. Dies kann ernste Konsequenzen haben, wie z.B. im Falle der Eingabe `rm t*`: eine der beiden Shells liest `rm *`, während die andere `t` liest.

Die Kosten für einen **fork**-Aufruf sind gewaltig. Das Kopieren des Vaters dient nur dazu, einige hundert Anweisungen auszuführen, bevor die Anweisungs- und Datensegmente reinitialisiert werden. Dabei wird möglicherweise ein großes Datensegment kopiert (auf das Anweisungssegment besteht üblicherweise nur Lesezugriff — es kann gemeinsam benützt werden). Dies klingt unrentabel und ist es auch. In einigen UNIX-Versionen mit virtueller Speicherverwaltung, wo Kopieren besonders teuer ist, wurde eine raffinierte Lösung, genannt *copy-on-write*, eingesetzt, die folgendermaßen funktioniert: bei einem **fork**-Aufruf teilen sich Vater und Sohn das Datensegment, das aus einer Anzahl von *Seiten* (*pages*) besteht. Solange eine Seite nicht modifiziert wird, richtet dieses abgekürzte Verfahren keinen Schaden an. Doch sobald der Vater oder der Sohn versucht, eine Seite zu schreiben, wird diese kopiert, damit jeder von ihnen seine eigene Version erhält.[13] Mit der richtigen Art von *paging*-Hardware (z.B. bei einer VAX) funktioniert dies einfach und wirtschaftlich. Da der **exec**-Aufruf fast immer sehr bald darauf folgt, müssen nur sehr wenige Seiten kopiert werden. Selbst wenn sie alle kopiert werden müssen, stehen wir nicht schlechter da — tatsächlich sind wir sogar besser dran, da wir den Sohn früher starten konnten. Erinnern Sie sich daran, daß das *copy-on-write*-Verfahren im Innern des Kerns stattfindet; die Bedeutung von **fork** wird dadurch nicht verändert, und abgesehen von der höheren Leistungsfähigkeit merken die Benutzer nichts davon. Die Methode folgt einer wichtigen Leitlinie für Systemprogrammierer: Optimiere ruhig das System, doch *laß die Benutzer in Frieden* — es ist nicht ihr Problem.

[13] In Berkeley 4.2 BSD gibt es einen Systemaufruf namens **vfork**, der die gleichen Ziele verfolgt, doch weniger elegant und nicht transparent. Der Vater wird solange gesperrt, bis der Sohn **exec** aufruft oder terminiert. Da dieser Systemaufruf sich von **fork** unterscheidet, ziehen Programme, die auf Berkeley 4.2 BSD portiert wurden, keinen Vorteil daraus. Selbstverständlich benützt die Berkeley-Shell (**csh**) den Systemaufruf **vfork**, was eine enorme Hilfe ist.

5.5 Der Systemaufruf "exit"

```
void exit(status)  /* Prozess terminieren */
   int status;     /* Exit-Status         */

/* kehrt nicht zurueck                    */
```

exit terminiert den aufrufenden Prozeß mit einem Status-Code, der dem am weitesten rechts stehenden Byte von **status** entspricht. Alle aktiven Filedeskriptoren werden geschlossen. Dank eines Tricks der C-Bücherei werden auch alle Datenströme der Standard-Bücherei der Ein- und Ausgabe geschlossen und die entsprechenden Puffer entleert, gerade so, als ob **fclose** aufgerufen worden wäre. Der Kern weiß davon jedoch nichts.

Sind zum Zeitpunkt des Aufrufs von **exit** noch irgendwelche Sohn-Prozesse am Leben, so werden diese davon nicht beeinträchtigt; doch da ihr Vater im Begriff ist zu sterben, ändert sich ihre Vater-Prozeßnummer in 1, also in die Nummer des **init**-Prozesses (Siehe Abschnitt 1.5).

Man sagt, daß **exit** der erste Systemaufruf ist, den jeder lernt. In der Tat sind sich die meisten C-Programmierer wahrscheinlich nicht bewußt, daß **exit** tatsächlich ein *Systemaufruf* und nicht nur ein Unterprogramm ist. Es gibt noch einen weiteren Unterschied: **exit** ist der einzige Systemaufruf, der *niemals* zurückkehrt.

Damit soll jedoch nicht gesagt werden, daß **exit** nicht auch falsch benutzt werden kann; tatsächlich ist dies oft der Fall, wenn **exit** wie folgt aufgerufen wird:

```
exit();
```

oder wenn man am Ende einer **main**-Funktion vorbeifällt. Als Status-Code des Prozesses wird Unsinn geliefert, der Schaden anrichten kann oder auch nicht. Am besten ist es natürlich, Dinge richtig zu erledigen. Per Konvention bedeutet der Status-Code 0, daß der Prozeß ordnungsgemäß terminiert ist, und ein Status-Code ungleich 0 (üblicherweise 1), daß ein Fehler aufgetreten ist. Es können jedoch alle 256 möglichen Codes in einem Byte verwendet werden.

Der Vater eines durch **exit** aus dem Leben scheidenden Prozesses erhält dessen Status-Code über einen **wait**-Systemaufruf.

5.6 Der Systemaufruf "wait"

```
int wait(statusp)  /* auf Sohn warten      */
   int * statusp;  /* Exit-Status          */

/* liefert Prozessnummer oder -1 bei Fehler */
```

Hat der Aufrufer irgendwelche Sohn-Prozesse erzeugt, so wartet **wait**, bis einer von ihnen terminiert. Der Aufrufer kann nicht bestimmen, auf welchen speziellen Sohn gewartet werden soll. **wait** liefert die Prozeßnummer des Sohns und speichert dessen Status-Code in der Integer-Zahl, auf die **statusp** verweist. Ist das Argument **NULL**, so wird der Status nicht gespeichert. **wait** kehrt auch dann zurück, wenn ein abgefangenes Signal empfangen wird oder wenn ein Sohn im sogenannten *Trace*-Modus anhält. Diese Dinge wollen wir aber für den Moment ignorieren; wir kommen in den Abschnitten 8.3 und 9.9 auf sie zurück.

Beachten Sie, daß es keine Enkel geben kann, wenn keine Söhne vorhanden sind: Stirbt ein Sohn, so werden seine Söhne von **init**, und nicht von deren Großvater adoptiert. Deshalb erhält ein Prozeß bei der Termination eines seiner Enkel kein Resultat von **wait**, da der Vater des Enkels immer noch am Leben ist und dieser beim Warten den Vorrang besitzt.

Ein Prozeß kann auch dann terminieren, wenn sein Vater nicht darauf wartet. Da der Kern sicherstellen muß, daß auf alle Prozesse gewartet wird, verwandelt sich ein derartiger Prozeß in einen sogenannten *Zombie*-Prozeß. Das Anweisungssegment, das Benutzer-Datensegment und das System-Datensegment des Prozesses werden weggeworfen, doch der Prozeß belegt immer noch einen Platz in der Prozeßtabelle im Kern.[14] Wird dann irgendwann einmal auch auf diesen Zombie gewartet, so verschwindet er endgültig.

Gibt es keine Söhne, sei es, daß es niemals welche gab oder weil frühere **wait**-Aufrufe ihre Prozeßnummern bereits zurückgeliefert haben, so liefert **wait** sofort –1 zurück und **errno** ist auf **ECHILD** gesetzt.

Es gibt keinen Grund dafür, daß auf alle Prozesse gewartet werden muß. Es ist völlig in Ordnung, wenn ein Prozeß einige Söhne erzeugt und danach seinen eigenen Geschäften nachgeht und die Söhne bis zu ihrer Terminierung laufen läßt.

Es gibt drei Möglichkeiten wie ein Prozeß terminiert: Er kann **exit** aufrufen, er kann ein tödlich wirkendes Signal empfangen oder das Betriebssystem kann abstürzen. Der über **statusp** zurückgelieferte Status-Code gibt an, welche der beiden ersten Ursachen die Termination bewirkt hat; im dritten Fall sind auch der Vater und der Kern tot, so daß der Status fraglich ist. Ist das am weitesten rechts stehende Byte des Status gleich Null, so bildet das Byte links davon das Argument des Sohns an **exit**. Ist das am weitesten rechts stehende Byte ungleich Null, so geben die sieben am weitesten rechts stehenden Bits die Nummer des Signals an, das den Sohn getötet hat. Ist das am weitesten links stehende Bit des am weitesten rechts stehenden Bytes 1, so wurde ein Speicherauszug produziert (in eine Datei namens **core** im Arbeitskatalog).

[14] Eine sehr treffende Lexikon-Definition von *Zombie* ist die folgende: "Eine Person auf der niedrigsten Intelligenzstufe, die den Eindruck des sogenannten wandelnden Toten erweckt." (*Webster's New Collegiate Dictionary, 6th ed.* [Springfield, Mass.: G & C. Merriam Co., 1960].)

Im folgenden zeigen wir Ihnen eine Funktion, die den Status-Code erläutert und eine angemessene Meldung ausgibt, wenn der Status-Code eine Signalnummer enthält (die Meldungen sind im Abschnitt 8.2 erklärt):

```
#define MAXSIG 19

void statusprt(pid, status)  /* Status-Code interpretieren */
   int pid, status;
{
   int code;
   static char *sigmsg[] =
   {
       "",
       "Hangup",
       "Interrupt",
       "Quit",
       "Illegal instruction",
       "Trace trap",
       "IOT instruction",
       "EMT instruction",
       "Floating point exception",
       "Kill",
       "Bus error",
       "Segmentation violation",
       "Bad arg to system call",
       "Write on pipe",
       "Alarm clock",
       "Terminate signal",
       "User signal 1",
       "User signal 2",
       "Death of child",
       "Power fail"
   };

   if (status != 0 && pid != 0)
       printf("Process %d: ", pid);
   if (lowbyte(status) == 0)
   {
       if ((code = highbyte(status)) != 0)
           printf("Exit code %d\n", code);
   }
   else
```

```
    {
        if ((code = status & 0177) <= MAXSIG)
            printf("%s", sigmsg[code]);
        else
            printf("Signal #%d", code);
        if ((status & 0200) == 0200)
            printf(" - core dumped");
        printf("\n");
    }
}
```

Die Makros **lowbyte** und **highbyte** wurden im Abschnitt 1.10 eingeführt.

Diese Funktion ähnelt derjenigen, die die Shell zur Ausgabe von Meldungen benutzt, mit denen alle Programmierer, abgesehen von den allerängstlichsten, vertraut sind. Eine solche Meldung lautet etwa:

```
Bus error - core dumped
```

Da wir nun im Besitz von **statusprt** sind, können wir **execut2** aus dem Abschnitt 5.4 verbessern:

```
static void execut3(argc, argv)  /* Kommando ausfuehren und warten */
    int argc;
    char * argv[];
{
    int status, code;

    switch (fork())
    {
        case -1:
            printf("Can't create new process\n");
            return;
        case 0:
            if (!EVupdate())
                fatal("can't update environment");
            execvp(argv[0], argv);
            fatal("can't execute");
        default:
            if (wait(&status) == -1)
                syserr("wait");
            statusprt(0, status);
    }
}
```

Wir haben uns die Mühe erspart, die von **wait** zurückgelieferte Prozeßnummer an **statusprt** weiterzureichen, da diese Shell immer nur einen Sohn gleichzeitig besitzt. Wir werden die Prozeßnummer ausnutzen, wenn wir in Kapitel 6 eine vollständigere Shell vorstellen.

Unser neuestes Versuchsmodell arbeitet tatsächlich so, wie es eine Shell tun sollte:

```
$ sh3
> date
> Thu Sep 6 17:15:39 MDT 1984
> who
marc console Sep 6 12:00
> echo This shell is much better!
This shell is much better!
> EOT
$
```

Die Buchstabenfolge *EOT* soll anzeigen, wo wir *Control*-D getippt haben.

5.7 Systemaufrufe zur Bestimmung von Identifikationsnummern

```
int getuid()   /* wirkliche Benutzernummer holen */

int getgid()   /* wirkliche Gruppennummer holen  */

int geteuid()  /* effektive Benutzernummer holen */

int getegid()  /* effektive Gruppennummer holen  */

int getpid()   /* Prozessnummer holen            */

int getpgrp()  /* Prozessgruppennummer holen     */

int getppid()  /* Vater-Prozessnummer holen      */
```

Jeder dieser Systemaufrufe liefert eine einzelne Nummer zurück, wie durch die Kommentare zu den Funktionsköpfen angedeutet. Einige von ihnen werden mit beliebiger Häufigkeit benutzt.

Die Prozeßnummer wird oft zur Konstruktion von Namen temporärer Dateien verwendet. Da jeder aktive Prozeß eine eindeutige Prozeßnummer besitzt, garantiert die Einbettung der Prozeßnummer in den Dateinamen Eindeutigkeit unter allen Prozessen. Das folgende Programmfragment ist typisch:

```
char file[10];
int tempfd;

sprintf(file, "TMP%d", getpid());
if ((tempfd = open(file, O_RDWR | O_CREAT | O_TRUNC, 0666)) == -1)
    syserr("open");
```

Mit dieser Technik können Sie jedoch nicht mehr als eine einzige temporäre Datei für einen bestimmten Prozeß erzeugen, da die Namen sonst alle gleich lauten würden. Wenn Sie ganz genau wissen, welche temporären Dateien benötigt werden, können Sie sie durch aufeinanderfolgende Buchstaben unterscheiden (z.B. **TMP23142a**, **TMP23142b** usw.). Um völlig sicherzugehen, können Sie mit **a** beginnen und bei **open** die **O_EXCL**-Flagge (siehe Abschnitt 2.6) benützen; mißlingt der Versuch, weil die Datei bereits existiert, so können Sie es mit **b** versuchen usw., bis Sie einen unbenutzten Dateinamen finden.

5.8 Die Systemaufrufe "setuid" und "setgid"

```
int setuid(uid)    /* Benutzernummer setzen */
    int uid;       /* Benutzernummer        */

/* liefert 0 bei Erfolg oder -1 bei Fehler */

int setgid(gid)    /* Gruppennummer setzen  */
    int gid;       /* Gruppennummer         */

/* liefert 0 bei Erfolg oder -1 bei Fehler */
```

Ist der Aufrufer der Super-User, so setzen die beiden Systemaufrufe die wirkliche und die effektive Benutzer- bzw. Gruppennummer auf das jeweilige Argument. Sie gestatten dem Super-User, die Identität eines anderen Benutzers anzunehmen.

Das wichtigste UNIX-Kommando, das diese Systemaufrufe benützt, ist **login**. Nach der Aufforderung zur Eingabe des Login-Namens und eines Paßworts prüft **login** die beiden Eingaben durch Nachschauen in der Datei **/etc/passwd**. Sind sie gültig, so werden **setgid** und **setuid** ausgeführt, um die wirkliche und effektive Gruppen- bzw. Benutzernummer entsprechend den Einträgen in **/etc/passwd** zu

setzen. Dann macht **login** den Login-Katalog zum Arbeitskatalog und benützt **exec** zur Ausführung der vorgesehenen Shell des Benutzers.

Gewöhnliche Benutzer dürfen **setuid** und **setgid** ebenfalls ausführen, doch nur zur Rückänderung der *effektiven* Benutzer- oder Gruppennummer in die entsprechende *wirkliche* Nummer. In diesem Fall muß das Argument zu **setuid** oder **setgid** mit der wirklichen Benutzer- oder Gruppennummer übereinstimmen. Das wird typischerweise dann benutzt, wenn ein Programm mit einer bestimmten effektiven Benutzernummer starten muß, um gewisse privilegierte Arbeiten zu verrichten, und danach für den restlichen Lebensweg des Prozesses zur wirklichen Nummer zurückgeschaltet wird.

5.9 Der Systemaufruf "setpgrp"

```
int setpgrp()  /* Prozessgruppennummer setzen */

/* liefert neue Prozessgruppennummer          */
```

Durch diesen Systemaufruf wird eine neue Prozeßgruppe gebildet. Beim Aufruf wird die Prozeßgruppennummer auf die Prozeßnummer gesetzt und der Prozeß aus seiner bisherigen Gruppe herausgenommen. Da alle Söhne des Prozesses die neue Prozeßgruppennummer erben, werden auch sie und alle ihre Nachfahren zu Mitgliedern der neuen Gruppe (solange nicht einer von ihnen ebenfalls eine neue Gruppe erzeugt).

Die Bedeutung von Prozeßgruppen liegt im Bereich der Signalverarbeitung (siehe dazu Abschnitt 1.5 und 8.5). Üblicherweise sind Prozesse der gleichen Gruppe irgendwie logisch miteinander verbunden. Zum Beispiel könnte ein aus mehreren Prozessen bestehendes Datenbank-Managersystem aus einem sogenannten *Master*-Prozeß und einigen untergeordneten Prozessen bestehen. Um aus einer Reihe von Datenbankprozessen eine Prozeßgruppe zu bilden, würde der *Master*-Prozeß einfach **setpgrp** aufrufen, bevor er seine Untergebenen erzeugt. Erinnern Sie sich daran, daß die Prozeßgruppennummer eines der Attribute ist, die ein Sohn von seinem Vater erbt.

5.10 Der Systemaufruf "chdir"

```
int chdir(path)    /* Arbeitskatalog aendern */
    char * path;   /* Pfadname               */

/* liefert 0 bei Erfolg oder -1 bei Fehler   */
```

Der Systemaufruf **chdir** ändert den Arbeitskatalog in den durch das Argument angegebenen Katalog. Für den neuen Arbeitskatalog wird Ausführungsrecht (Suchrecht), nicht jedoch Lese- oder Schreibrecht benötigt.

Die wichtigste Anwendung von **chdir** stellt natürlich das **cd**-Kommando dar, das in die Shell eingebaut ist. Wäre es nicht eingebaut, sondern würde es durch einen Sohn-Prozeß bewerkstelligt werden, wäre es wirkungslos, weil ein von einem Sohn-Prozeß verändertes Attribut (in diesem Fall der Arbeitskatalog) nicht an den Vater zurückgereicht werden kann.

5.11 Der Systemaufruf "chroot"

```
int chroot(path)     /* Wurzelkatalog aendern */
    char * path;     /* Pfadname              */

/* liefert 0 bei Erfolg oder -1 bei Fehler   */
```

Dieser Systemaufruf ändert den Wurzelkatalog (der Katalog mit dem Namen /) eines Prozesses. Nur der Super-User darf diesen Systemaufruf ausführen. **chroot** ist nützlich, wenn ein Kommando mit eingebauten Pfadnamen (wie **/etc/passwd**) nicht auf den üblichen Dateien ablaufen soll. Ein neuer Baum kann an irgendeiner geeigneten Stelle angelegt, die Wurzel verändert und danach das Kommando ausgeführt werden. Dies ist vor allem zum Testen von neuen Kommandos nützlich, bevor diese installiert werden.

Die Beschränkung auf den Super-User hindert gewöhnliche Benutzer daran, Systemdateien zu fälschen und die Wurzel zu ändern, um ein Kommando aufs Glatteis zu führen. Anderenfalls könnte man nach folgendem Schema Super-User werden (einige wenige Einzelheiten wurden weggelassen):

1. Legen Sie irgendwo einen Katalog namens **etc** an, **/usr/marc/etc** genügt.
2. Kopieren Sie **/etc/passwd** in diesen Katalog.
3. Editieren Sie die Kopie, um das Paßwort von **root** (Login-Name des Super-Users) zu löschen.
4. Schreiben Sie ein Programm, das **chroot** aufruft, um die Wurzel in **/usr/marc** zu ändern, und das danach **login** ausführt.
5. Lassen Sie das Programm laufen und melden Sie sich als Super-User an.

Diese Methode funktioniert glücklicherweise nicht, da Sie Super-User sein müssen, um **chroot** verwenden zu können.[15]

[15] Es klappt außerdem nicht, da **login** für Sie erreichbar sein muß. A.d.Ü.

Vielleicht glauben Sie, daß man dies auch dadurch verhindern kann, daß **login** anders programmiert wird, etwa so, daß man den Aufruf

```
chroot("/");
```

einfügt, um die Wurzel wieder auf ihren angestammten Platz zu setzen. Damit liegen Sie aber falsch. Ein solcher Aufruf bewirkt absolut nichts, weil **chroot** auch auf die neue Wurzel angewendet wird. Tatsächlich gibt es keine Möglichkeit, zur alten Wurzel zurückzukehren, wenn man sie einmal verändert hat. Wir wissen zwar, daß die normale Wurzel die Inumber 2 besitzt, doch ist es nicht möglich, diese Tatsache auszunutzen. Auch können wir uns von der Wurzel aus nicht nach oben bewegen, da der Pfad .. relativ zur Wurzel nur wieder zurück zur Wurzel führt.

5.12 Der Systemaufruf "nice"

```
int nice(incr)     /* nice-Wert aendern          */
    int incr;      /* Inkrement                  */

/* liefert neuen nice-Wert oder -1 bei Fehler */
```

Jeder Prozeß besitzt einen sogenannten *nice*-Wert, den wir bisher immer mit *Priorität* bezeichnet haben.[16] Der Systemaufruf **nice** addiert **incr** zum *nice*-Wert. Der resultierende *nice*-Wert muß zwischen 0 und 39 (jeweils einschließlich) liegen; ergibt sich ein unzulässiger Wert, so wird der am nächsten liegende gültige Wert verwendet. Höhere *nice*-Werte bedeuten schlechtere Versorgungsleistung. Prozesse starten mit dem *nice*-Wert 20. Nur der Super-User kann den *nice*-Wert erniedrigen, um überdurchschnittlichen Service zu erlangen.

nice liefert in Wirklichkeit den um 20 verringerten *nice*-Wert zurück, so daß der Resultatwert im Bereich von –20 bis 19 liegt. Der Resultatwert ist jedoch nur selten von großem Nutzen. Dies ist auch gut so, da der neue *nice*-Wert 19 von einem Resultat, das einen Fehler anzeigt, nicht unterscheidbar ist (19 – 20 = –1). Die Tatsache, daß dieser Fehler über so viele Jahre hinweg nicht behoben wurde, wenn nicht sogar unentdeckt geblieben ist, zeigt, wie wenig sich die meisten UNIX-Systemprogrammierer um fehleranzeigende Resultate von weniger bedeutenden Systemaufrufen wie **nice** kümmern.

Die meisten UNIX-Benutzer sind mit dem **nice**-*Kommando* vertraut, das ein Programm mit niedrigerer Priorität laufen läßt (oder mit höherer Priorität, falls es vom Super-User aufgerufen wird). Hier ist unsere Fassung dieses Kommandos:

[16] Wir führen den merkwürdigen Begriff *nice*-Wert ein, da ihn das ps-Kommando in seiner Ausgabe verwendet. In Wirklichkeit ist die *Priorität* eine ganz andere Zahl, die sowohl durch den *nice*-Wert als auch durch andere Attribute beeinflußt wird.

```
#define USAGE "usage: nice [-num] command\n"

main(argc, argv)  /* nice-Kommando */
    int argc;
    char * argv[];
    {
        int incr, cmdarg;

        if (argc < 2)
        {
            fprintf(stderr, USAGE);
            exit(1);
        }
        if (argv[1][0] == '-')
        {
            incr = atoi(&argv[1][1]);
            cmdarg = 2;
        }
        else
        {
            incr = 10;
            cmdarg = 1;
        }
        if (cmdarg >= argc)
        {
            fprintf(stderr, USAGE);
            exit(1);
        }
        nice(incr);  /* return-Wert ignorieren */
        execvp(argv[cmdarg], &argv[cmdarg]);
        syserr("execvp");
}
```

Beachten Sie, daß wir **argv** beim Aufruf von **execvp** wiederverwenden. Die Variable **cmdarg** enthält den Index des Kommandonamens (entweder 1 oder 2, je nachdem, ob ein Inkrement angegeben wurde). Der bei **cmdarg** beginnende Teil von **argv** ist der Teil, der weitergereicht werden muß. Möglicherweise stellen Sie dies den umfangreicheren Maßnahmen gegenüber, die für **execvp2** aus dem Abschnitt 5.3 nötig waren. Dort mußten wir jedoch ein Argument *einfügen*, was erforderte, daß wir den ganzen Vektor umkopieren mußten.

Beachten Sie weiterhin, daß wir **execvp** ohne **fork** verwendet haben. Haben wir die Priorität einmal geändert, so besteht keine Notwendigkeit mehr zu warten. Das

nohup-Kommando verwendet eine ähnliche Methode. Das **time**-Kommando dagegen muß warten, um die Ausführungszeiten zu drucken, so daß es zur Ausführung des Kommandos einen Sohn-Prozeß erzeugt.

5.13 Portabilität

In Version 7 und in 4.2 BSD sind die Systemaufrufe **execvp** und **execlp** nicht enthalten, und in Version 7 gibt es außerdem **chroot** nicht. Dies beeinträchtigt die Portabilität nicht sonderlich, da die beiden **exec**-Varianten als Unterprogramme geschrieben werden können (siehe **execvp2** in Abschnitt 5.3) und **chroot** nur selten benutzt wird (und praktisch niemals in einem Anwendungsprogramm enthalten ist).

In Version 7 liefert **nice** kein Resultat. Dies ist auch gut so, da die neuere Form von **nice** ohnehin nur einen nutzlosen Resultatwert liefert (siehe Abschnitt 5.12).

Vieles in diesem Kapitel ist portabel, da es uralt ist. Diese Systemaufrufe liegen mitten im Herzen von UNIX und sind mit Ausnahme der wenigen aufgeführten Fälle über viele Jahre hinweg unverändert geblieben. Es ist bedauerlich, daß Portabilität auf hohem Alter anstatt auf einer gewissenhaft verwirklichten Planungsphilosophie beruht, doch wir akzeptieren Portabilität, egal woher sie kommt.

Aufgaben

5.1 Modifizieren Sie die Funktionen zur Manipulation des Environments aus Abschnitt 5.2, so daß exportierte Variablen genauso behandelt werden wie in der Standard-Shell (**sh**). Das heißt, ein veränderter Wert einer Variablen wird nur dann exportiert, wenn diese ausdrücklich in einem **export**-Kommando aufgeführt ist.

5.2 Modifizieren Sie die Funktionen zur Manipulation des Environments, so daß ein Hash-Verfahren anstelle des linearen Suchverfahrens eingesetzt wird.

5.3 Schreiben Sie ein Programm, das seine Argumente nach Zuweisungen der Form *variable=value* durchsucht, das Environment entsprechend ändert und dann das Programm ausführt, das durch das erste Argument bestimmt wird, das keine Zuweisung darstellt. Alle anderen Argumente, die ebenfalls keine Zuweisung darstellen, sollen Argumente des aufzurufenden Programms werden. Benützen Sie hierfür nicht **fork**.

5.4 Schreiben Sie eine Funktion **execlp2** analog zur Funktion **execvp2** aus Abschnitt 5.3.[17]

[17] Hinweis: Betrachten Sie **vararg.h**. A.d.Ü.

5.5 Entwerfen Sie einen Plan, mit dem man die durch **fork** verbrauchte CPU-Zeit messen kann. (Vielleicht sollten Sie hierfür die Zeitmeßfunktionen aus Abschnitt 9.10 verwenden). Falls möglich, probieren Sie dies unter verschiedenen UNIX-Versionen und auch auf verschiedenen Computermodellen aus. Falls Sie Zugang zu Berkeley-UNIX haben, vergleichen Sie die Zeiten für **fork** und **vfork**.

5.6 Untersuchen Sie, welche Systemaufrufe in anderen Betriebssystemen wie VMS (DEC) und MVS (IBM) als Äquivalent zu **exec** und **fork** vorhanden sind. Vergleichen Sie ihre Fähigkeiten und zählen Sie ihre Vor- und Nachteile auf.

Kapitel 6: Elementare Prozeßkommunikation

6.1 Einführung

Da wir nun wissen, wie man Prozesse erzeugt, wollen wir sie so verknüpfen, daß sie miteinander kommunizieren können. In diesem Kapitel werden wir das mit Hilfe von Pipes realisieren und dabei Techniken anwenden, die auf allen UNIX-Versionen funktionieren. Im nächsten Kapitel werden wir die Prozeßkommunikation weiter erforschen, indem wir Einrichtungen benutzen, die es nur unter System III, System V und Xenix 3 gibt, wie etwa FIFO-Dateien und *Messages*.

Pipes sind den meisten UNIX-Benutzern im Rahmen der Shell wohlbekannt. Um zum Beispiel eine sortierte Liste der angemeldeten Benutzer auszudrucken, kann man folgende Kommandozeile eingeben:

```
who | sort | pr
```

Hier gibt es drei Prozesse, die durch zwei Pipes miteinander verbunden sind. Daten fließen allerdings nur in eine Richtung — von **who** über **sort** an **pr**. Es ist aber auch möglich, bidirektionale *Pipelines* aufzubauen (von Prozeß A nach B, und von B zurück nach A), sowie Pipelines in einem Ring (von A nach B nach C und wieder nach A). Beides geht, indem man Systemaufrufe benützt. Die Shell jedoch bietet keine Vokabel zum Benutzen dieser kunstvolleren Einrichtungen, so daß sie den meisten UNIX-Benutzern unbekannt sind.[1]

Wir beginnen mit ein paar einfachen Beispielen: Prozesse, die durch eine in einer Richtung durchlässige Pipe verbunden sind. Dann werden wir die einfache Shell verbessern, die wir in Kapitel 5 entwickelt haben. Unsere neue Shell wird vollständig genug sein, daß sie wirklich den Namen Shell verdient — sie wird Pipelines, Hintergrundprozesse, Ein- und Ausgabeumlenkung und in Anführungszeichen gesetzte

[1] Diese Anordnungen können zwar mit FIFO-Dateien erzielt werden, aber die Shell ist daran unbeteiligt — sie denkt, es wären Dateien.

Argumente beherrschen. Was ihr noch fehlen wird ist die Generierung von Dateinamen (`ls t*.?`) und Kontrollstrukturen. Zuletzt zeigen wir, wie man Prozesse mit bidirektionalen Pipes verbindet, und legen die *Deadlock*-Probleme dar, die sich hier ergeben können.

6.2 Der Systemaufruf "pipe"

```
int pipe(pfd)       /* eine Pipe einrichten */
    int pfd[2];     /* Filedeskriptoren     */

/* liefert 0 bei Erfolg oder -1 bei Fehler */
```

Der Systemaufruf **pipe** erzeugt eine *Pipe* — einen Kanal zur Prozeßkommunikation, der durch zwei Filedeskriptoren repräsentiert wird, die im Vektor **pfd** zurückgeliefert werden. Man schreibt Daten in die Pipe über **pfd[1]** und liest Daten aus der Pipe über **pfd[0]**. Einige Ein- und Ausgabe-Systemaufrufe verhalten sich bei Filedeskriptoren dieser Art anders als bei gewöhnlichen Dateien, und manche tun sogar gar nichts. Die Einzelheiten dazu sind wie folgt:

write Daten, die in eine Pipe geschrieben werden, werden sequentiell in Reihenfolge des Eintreffens abgelegt. Wenn eine Pipe voll ist, blockiert **write** normalerweise solange, bis durch **read** genügend alte Daten entfernt sind. Es gibt keine teilweisen Schreibvorgänge, immer wird der ganze **write**-Aufruf abgeschlossen. Die Kapazität einer Pipe variiert mit der UNIX-Implementierung, jedoch sind es immer mindestens 4096 Bytes. Falls über den Aufruf **fcntl** die **O_NDELAY**-Flagge gesetzt wird, so blockiert **write** bei einer vollen Pipe nicht, sondern liefert als Resultat den Wert 0. Die einzige Möglichkeit, ein *end-of-file* in eine Pipe zu "schreiben", ist, den Filedeskriptor für den Schreibzugriff zu schließen.

read Daten werden in der Reihenfolge gelesen, in der sie eintrafen, d.h. in der sie geschrieben wurden. Einmal gelesen, können die Daten kein zweites Mal gelesen oder gar in die Pipe zurückgestellt werden. Wenn die Pipe leer ist, blockiert **read** normalerweise solange, bis mindestens ein Byte verfügbar ist, es sei denn, der Filedeskriptor für den Schreibzugriff wird unterdessen geschlossen — dann liefert **read** eine 0 (der übliche Hinweis auf Dateiende). Weiterhin wird das dritte Argument von **read** (die Zahl der zu lesenden Bytes) nicht notwendigerweise befriedigt — nur so viele Bytes, wie gerade vorhanden sind, werden gelesen und die entsprechende Anzahl zurückgegeben.

Natürlich wird nie eine höhere Zahl als angegeben geliefert — nicht gelesene Bytes harren auf das nächste **read**. Wenn die **O_NDELAY**-Flagge gesetzt ist, liefert **read** auf eine leere Pipe den Wert 0. Auch dieser Aufruf leidet an der gleichen Zweideutigkeit wie **read** bei einer Terminalverbindung (siehe Abschnitt 4.3): Der Wert 0 bedeutet auch Dateiende.

close Dieser Systemaufruf hat bei einer Pipe mehr Auswirkungen als bei einer Datei. Wenn der Filedeskriptor für Schreibzugriff, also **pfd[1]**, geschlossen wird, wird er nicht nur zur weiteren Benutzung freigegeben, sondern der Lese-Prozeß erhält dann auch ein *end-of-file*. Wenn dagegen der Filedeskriptor für den Lesezugriff geschlossen wird, verursacht ein **write** auf den anderen Filedeskriptor einen Fehler. Hierbei wird normalerweise auch ein Signal erzeugt (siehe Abschnitt 8.2).

fcntl Dieser Aufruf setzt oder löscht die **O_NDELAY**-Flagge, deren Auswirkung unter **read** und **write** beschrieben ist.

fstat Nicht sehr nützlich bei Pipes. Die Dateigröße, die zurückgeliefert wird, ist die Anzahl Bytes, die sich gerade in der Pipe befinden, was allerdings selten von großem Nutzen ist. Über die Verweiszahl kann eine Pipe von einer Datei unterschieden werden; bei einer Pipe ist diese 0, da dies der einzige Fall ist, bei dem ein Filedeskriptor mit etwas verbunden ist, was selbst nicht in einem Katalog vorkommt. Diese Unterscheidung kann bei Ein- und Ausgabe-Funktionen nützlich sein, die Pipes besonders behandeln wollen.

dup Dieser Systemaufruf wird in Abschnitt 6.3 erklärt.

open Keine Verwendung bei Pipes.

creat Keine Verwendung bei Pipes.

lseek Keine Verwendung bei Pipes. Das bedeutet, daß man eine Pipe, die mehrere Portionen Daten enthält, nicht durchsuchen kann. Es ist wie bei einer Zahnpastatube: Um den Inhalt zu untersuchen, muß man ihn erst herausdrücken; danach gibt es keine Möglichkeit, ihn wieder hineinzubringen. Das ist übrigens einer der Gründe, warum Pipes für Anwendungsprogramme, die Meldungen zwischen Prozessen austauschen, umständlich sind.

Pipes benützen das Blockdepot genau wie normale Dateien. Demnach ist der Nutzen, Pipes blockweise (normalerweise in Stücken zu 512 Bytes) zu lesen und zu schreiben, genauso groß. Eine einzelne Ausführung von **write** ist atomar, d.h. wenn 512 Bytes mit einem einzigen Aufruf geschrieben werden, so wird auch der entsprechende **read**-Aufruf 512 Bytes lesen (sofern er so viele angefordert hat). **read** wird dann niemals mit weniger als dem ganzen Block zurückkehren. Falls jedoch der Sender nicht komplette Blöcke schreibt, der Empfänger aber versucht, solche zu le-

sen, so kann er trotzdem unvollständige Blöcke erhalten. Dies kann nicht passieren, wenn der Sender schneller ist als der Empfänger, da dann der Sender in der Lage ist, die Pipe mit einem ganzen Block zu füllen, bevor der Empfänger überhaupt irgend etwas lesen kann. Trotzdem ist es am besten, Lese- und Schreibzugriffe bei Pipes zu puffern, und genau das tut ja die Standard-Bücherei der Ein- und Ausgabe automatisch.

Betrachtet man einen einzelnen Prozeß, welchen Sinn hat dann eine Pipe? Natürlich gar keinen. Zur Veranschaulichung folgt nun trotzdem ein dummes Beispiel:

```
pipetest()  /* Lesen und Schreiben einer Pipe */
{
    int pfd[2], nread;
    char s[100];

    if (pipe(pfd) == -1)
        syserr("pipe");
    if (write(pfd[1], "hello", 6) == -1)
        syserr("write");
    switch (nread = read(pfd[0], s, sizeof(s)))
    {
        case -1:
            syserr("read");
        case 0:
            fatal("EOF");
        default:
            printf("read %d bytes: %s\n", nread, s);
    }
}
```

Und die dazugehörige Ausgabe:

```
read 6 bytes: hello
```

Wir konnten sechs Bytes in die Pipe schreiben, ohne uns Sorgen um ihren Füllgrad machen zu müssen, da sie mindestens eine Kapazität von 4096 Bytes besitzt. Falls wir jedoch mehr schreiben würden, und die Pipe schließlich voll wäre, so würde **write** blockieren, bis **read** die Pipe wieder etwas geleert hätte. Dies würde aber nie passieren, da das Programm nie den **read**-Aufruf erreichen könnte. Wir würden in einer Situation festsitzen, die *Deadlock*[2] genannt wird. Deshalb muß man beim Benutzen von Pipes vorsichtig sein, um gerade solche Deadlock-Situationen zu vermeiden. Allerdings brauchen Sie sich nicht übermäßig Sorgen zu machen: Wenn

[2] "die tödliche Umklammerung". A.d.Ü.

Sie Prozesse mit einzelnen, nicht bidirektionalen Pipes verbinden (so wie es die Shell macht), so sind Deadlock-Situationen unmöglich (zumindest mit Pipes als Ursache). Nur die etwas exotischeren Konstruktionen können Deadlock-Situationen verursachen.

Nehmen wir nun einmal an, wir hätten zwei Prozesse. Wie können wir diese miteinander verbinden, so daß der eine aus einer Pipe das lesen kann, was der andere hineinschreibt? Die Antwort ist ganz einfach: gar nicht. Wenn die Prozesse einmal erzeugt sind, können sie nicht mehr miteinander verbunden werden, da es für den Prozeß, der die Pipe erzeugt, keine Möglichkeit gibt, einen Filedeskriptor an den anderen Prozeß zu übergeben. Er kann zwar die Nummer des Filedeskriptors übergeben, aber diese hat in dem anderen Prozeß keine Gültigkeit. Falls wir aber in einem Prozeß eine Pipe generieren, *bevor* der andere Prozeß erzeugt wird, so werden die Filedeskriptoren der Pipe vererbt und besitzen dann in beiden Prozessen Gültigkeit. Demzufolge können zwei miteinander kommunizierende Prozesse etwa aus Vater und Sohn bestehen, oder aus zwei Geschwistern, oder aus Großvater und Enkel, und so weiter. Jedenfalls müssen sie miteinander verwandt sein, und die Pipe muß bei der Geburt weitergereicht werden. In der Praxis kann das eine schwerwiegende Beschränkung darstellen, da es beim Ableben eines Prozesses keine Möglichkeit gibt, ihn wieder zum Leben zu erwecken und ihn erneut mit seinen Pipes zu verknüpfen — somit müssen auch die Überlebenden vernichtet werden und die ganze Familie muß neu erzeugt werden.

Im folgenden Beispiel generiert ein Prozeß (der **pwrite** ausführt) eine Pipe, erzeugt einen Sohn-Prozeß (der **pread** ausführt), der die Pipe erbt, und schreibt dann einige Daten zum Lesen für den Sohn-Prozeß in die Pipe. Obwohl der Sohn-Prozeß den nötigen Filedeskriptor für den Lesezugriff geerbt hat, weiß er dessen Nummer noch nicht (es gibt 20 Möglichkeiten), deshalb bekommt er diese Nummer als Argument mitgeliefert.

```
pwrite()  /* Daten werden fuer pread in die Pipe geschrieben */
{
    int pfd[2];
    char fdstr[10];

    if (pipe(pfd) == -1)
        syserr("pipe");
    switch (fork())
    {
        case -1:
            syserr("fork");
        case 0:
            if (close(pfd[1]) == -1)
                syserr("close");
```

```
            sprintf(fdstr, "%d", pfd[0]);
            execlp("./pread", "pread", fdstr, NULL);
            syserr("execlp");
        }
        if (close(pfd[0]) == -1)
            syserr("close2");
        if (write(pfd[1], "hello", 6) == -1)
            syserr("write");
    }
```

Hier nun das Programm für den Sohn-Prozeß:

```
main(argc, argv)  /* Daten von pwrite lesen */
    int argc;
    char *argv[];
{
    int fd, nread;
    char s[100];

    fd = atoi(argv[1]);
    printf("reading file descriptor %d\n", fd);
    switch (nread = read(fd, s, sizeof(s)))
    {
        case -1:
            syserr("read");
        case 0:
            fatal("EOF");
        default:
            printf("read %d bytes: %s\n", nread, s);
    }
}
```

Und hier ist die Ausgabe:

```
reading file descriptor 3
read 6 bytes: hello
```

Zu diesem Beispiel sind vielleicht einige Anmerkungen nötig: Da der Sohn aus der Pipe nur liest, nicht aber in sie schreibt, schließt er das schreibbare Ende der Pipe (**pfd[1]**) sofort (die Anweisung nach **case 0** bei **pwrite**), um Filedeskriptoren zu sparen. **sprintf** wandelt den Filedeskriptor zum Lesen von einer Integer in eine Zeichenkette um (UNIX verfügt über die Standardfunktion **atoi**, nicht aber über **itoa**), so daß diese als Argument an ein Programm übergeben werden kann. Da wir wissen, daß sich das Sohn-Programm im aktuellen Katalog befindet, können wir seinen

Pfad als **./pread** angeben, um **execlp** unnötige Suche zu ersparen. Das spart Zeit; was aber noch wichtiger ist, es verhindert eine unbeabsichtigte Ausführung eines anderen **pread**-Programms (wer weiß, was in **/usr/bin** alles steht). Dies hätten wir auch durch **execl** erreichen können, da dieser Aufruf keine Suche bewirkt, aber wir ziehen es vor, nur zwei Varianten von **exec** zu benutzen (die andere heißt **execvp**).

Unterdessen hat der Vater keine Verwendung für das lesbare Ende der Pipe, so schließt er **pfd[0]**. Dann schreibt er sechs Bytes in die Pipe und es steht ihm dann frei, sich mit anderen Dingen zu beschäftigen. Wir geben uns nicht damit ab, auf das Ende des Sohns zu warten; nachdem wir dem Sohn Daten geschickt haben, kümmern wir uns nicht mehr um ihn.

Der Sohn führt **pread** aus, das nun gerade sein Argument in eine Integer zurückverwandelt und von diesem Filedeskriptor liest. Weiterhin gibt er einige Meldungen aus, damit man sieht, daß die Sache auch wirklich geklappt hat.

Hier ist noch einmal das allgemeine Vorgehen, um zwei Prozesse über eine Pipe miteinander zu verknüpfen:

1. Die Pipe generieren.
2. **fork** ausführen, um den lesenden Sohn-Prozeß zu erzeugen.
3. Im Sohn-Prozeß das schreibbare Ende der Pipe schließen und andere nötige Vorbereitungen treffen (welche das sind, werden wir in späteren Beispielen sehen).
4. Im Sohn-Prozeß das gewünschte Programm ausführen.
5. Im Vater-Prozeß das lesbare Ende der Pipe schließen.
6. Wenn ein zweiter Sohn in die Pipe schreiben soll, diesen erzeugen, die nötigen Vorbereitungen treffen und sein Programm ausführen. Falls der Vater schreiben soll, dann eben dessen Programm weiter ausführen und schreiben.

Alle unsere Beispiele von nicht bidirektionalen Pipes werden sich an dieses Muster halten.

Wenn Sie an Abschnitt 5.3. zurückdenken, so versprachen wir dort eine Rechtfertigung der Trennung von **fork** und **exec**. Warum nicht ein einziger Systemaufruf, der beide Aufgaben erledigt und somit den Aufwand von **fork** spart? Nun wollen wir unser Versprechen einlösen: Die zwei werden gerade deshalb getrennt, um die Ausführung von Schritt 3 von oben zu ermöglichen. Wir haben schon einen Teil der Arbeit entdeckt, die zwischen **fork** und **exec** zu tun ist (**pfd[1]** schließen), und im Verlauf dieses Kapitels werden wir noch mehr sehen. Man kann diese Arbeit nicht vor dem **fork**-Aufruf erledigen, da wir nicht wollen, daß der Vater-Prozeß beeinflußt wird (der ja das schreibbare Ende der Pipe nicht schließen darf). Wir wollen auch nicht, daß das Programm des Sohns diese Arbeit macht, denn Programme sollten

nichts davon bemerken, wie sie aufgerufen wurden und wie ihre Ein- und Ausgaben verbunden sind — dies ist ein Grundstein der UNIX-Philosophie. Deshalb erledigen wir die Arbeit an der bestmöglichen Stelle: im Sohn, der den vom Vater kopierten Code ausführt. Ein weiterer Nutzen ist, daß der ganze Programmtext, der die Prozeßverbindung beschreibt, an einer Stelle steht. Das erleichtert Fehlersuche und Änderungen.

Hartnäckig wie wir sind, stellen wir uns trotzdem noch folgende Frage: Da es nur wenige typische Möglichkeiten gibt, wie Prozesse miteinander verbunden sind, wieso gibt es dann nicht einen einzigen fork-exec-Systemaufruf mit verschiedenen Optionen für die Prozeßverbindung? Schließlich gibt es noch andere Systemaufrufe mit einer Fülle von Optionen, wieso dann nicht auch hier? Die Antwort ist einfach die: Die ursprünglichen Entwickler von UNIX, Thompson und Ritchie, haben überdurchschnittlich guten Geschmack. **fork** und **exec** sind einfach, sie erlauben dem Benutzer aber trotzdem ein reichhaltiges Spektrum an gängigen Verknüpfungen — weshalb sollte man dann einen neuen Systemaufruf zusammenmanschen?

Das Programm **pread** wurde extra dafür entworfen, von einem Filedeskriptor zu lesen, dessen Nummer übergeben wurde. Das ist allerdings sehr ungewöhnlich — kein Standard-UNIX-Kommando funktioniert so. Viele Programme lesen zwar von einem bestimmten Filedeskriptor, von dem sie annehmen, daß er bereits aktiv ist, aber dieser hat die feste Nummer 0 (die Standard-Eingabe). Er muß also nicht als Argument übergeben werden. Entsprechend sind auch viele Programme zum Schreiben auf den Filedeskriptor mit der Nummer 1 (die Standard-Ausgabe) konzipiert. Um Kommandos wie in der Shell miteinander zu verbinden, müßten wir **pipe** irgendwie dazu bringen, bestimmte Filedeskriptoren im Vektor **pfd** zurückzuliefern: 0 für das lesbare und 1 für das schreibbare Ende. Leider bietet **pipe** keine solche Möglichkeit. Wir könnten zwar versuchen, vor dem **pipe**-Aufruf die Filedeskriptoren 0 und 1 zu schließen, um diese verfügbar zu machen, doch leider funktioniert das nicht. Das liegt daran, daß das Handbuch keinerlei Aussage darüber macht, welche Filedeskriptoren **pipe** benützt. Außerdem können wir es uns in der Regel nicht leisten, unsere Standard-Ausgabe und Standard-Eingabe zu opfern, nur um eine Pipe zu machen. Wie funktioniert es aber dann? Indem wir **dup** benutzen.

6.3 Der Systemaufruf "dup"

```
int dup(fd)             /* Verdoppeln eines Filedeskriptors      */
    int fd;             /* der bestehende Filedeskriptor         */

/* liefert einen neuen Filedeskriptor oder -1 bei Fehler         */
```

dup verdoppelt einen bestehenden Filedeskriptor und liefert als Resultat einen neuen Filedeskriptor, der mit der gleichen Datei oder Pipe verbunden ist. Beide Filedeskriptoren teilen sich dann den gleichen Positionszeiger, genau wie wenn ein

geerbter Filedeskriptor beim Sohn den Positionszeiger mit dem entsprechenden Filedeskriptor des Vaters teilt. Der Aufruf geht schief, falls das Argument falsch ist (nämlich nicht aktiv) oder wenn bereits 20 Filedeskriptoren aktiv sind.

Da der Positionszeiger also von beiden gemeinsam genutzt wird, hat die Existenz eines zweiten Filedeskriptors nur einen Sinn: Einen weiteren Filedeskriptor mit einer anderen Nummer zu haben, der damit eventuell den Anforderungen des Aufrufers besser genügt. Hier ist garantiert, daß der Filedeskriptor mit der kleinsten freien Nummer gewählt wird. Diese Regel kann dazu benützt werden, einen Filedeskriptor mit der Nummer 0 zu erhalten, der das lesbare Ende einer Pipe darstellt: Schließen Sie einfach den Filedeskriptor 0 und führen Sie **dup** mit dem lesbaren Ende der Pipe aus. Da 0 die kleinste mögliche Nummer ist, und da wir diese gerade verfügbar gemacht haben, wird **dup** einen Filedeskriptor mit dieser Nummer liefern. Falls wir dann durch **exec** ein Programm ausführen, das den Filedeskriptor 0 liest, so haben wir das Programm dazu gebracht, aus der Pipe zu lesen. Ein ähnlicher Algorithmus kann benutzt werden, um den Filedeskriptor 1 zum schreibbaren Ende einer Pipe zu machen. Tatsächlich kann man über **close** und **dup** einen Filedeskriptor mit jeder gewünschten Nummer erhalten. Normalerweise werden wir allerdings nur die Filedeskriptoren 0 und 1 so manipulieren.

Anstelle von **dup** kann man auch **fcntl** mit dem Argument **F_DUPFD** benutzen (siehe Abschnitt 3.13). Der zurückgelieferte Filedeskriptor ist ein Duplikat des ersten Arguments, allerdings wird garantiert, daß dieser größer oder gleich dem dritten Argument ist. Um zum Beispiel Filedeskriptor 6 als Kopie von Filedeskriptor **fd** zu erzeugen, wäre folgendes nötig:

```
(void) close(6);  /* Fehler ignorieren */
if ((newfd = fcntl(fd, F_DUPFD, 6)) == -1)
    syserr("fcntl");
```

Wir schlossen Filedeskriptor 6, um ihn verfügbar zu machen, für den Fall, daß er aktiv war.

Ohne **fcntl** wäre diese Aufgabe nicht so leicht zu lösen gewesen. Außer daß wir Filedeskriptor 6 verfügbar machen müßten, müßten wir sichergehen, daß die Filedeskriptoren 0 bis 5 nicht verfügbar sind, da sonst einer von diesen durch **dup** ausgewählt würde. Das genaue Vorgehen wäre wie folgt:

1. So oft **dup** aufrufen, bis die gewünschte Nummer geliefert wird.

2. Alle bis dahin irrtümlich duplizierten Filedeskriptoren schließen.

Die Arbeit, die durch **fcntl** gespart wird, ist im Normalfall unerheblich, da 0 und 1 sehr niedrige Nummern sind und niemals verfügbar sind, so daß ein **close**-Aufruf und ein **dup**-Aufruf genügt. Ein Vorteil, wenn man **fcntl** nicht benutzt: In den meisten UNIX-Systemen, die auf Version 7 basieren, gibt es **fcntl** gar nicht.

Wir könnten nun wieder die gleiche Frage wie vorhin stellen: Wenn es nun **fcntl** gibt, weshalb ist **dup** dann nicht nur ein Unterprogramm? Die Antwort ist wieder einmal leicht: aus Tradition.

Um **dup** zu illustrieren, folgt nun ein Beispiel. Erst wird eine Pipe erzeugt, danach ein Sohn, der diese lesen soll. Dann wird die Standard-Eingabe des Sohns auf das lesbare Ende der Pipe umgebogen, und zu guter Letzt wird das **cat**-Kommando aufgerufen, um die Pipe zu lesen (da wir nun die Standard-Eingabe lesen können, benötigen wir kein spezielles Programm wie **pread** mehr):

```
pwrite2()  /* Daten ueber eine Pipe zu cat umlenken */
{
    int pfd[2];

    if (pipe(pfd) == -1)
        syserr("pipe");
    switch (fork())
    {
        case -1:
            syserr("fork");
        case 0:
            if (close(0) == -1)
                syserr("close");
            if (dup(pfd[0]) != 0)
                fatal("dup");
            if (close(pfd[0]) == -1 || close(pfd[1]) == -1)
                syserr("close2");
            execlp("cat", "cat", NULL);
            syserr("execl");
    }
    if (close(pfd[0]) == -1)
        syserr("close3");
    if (write(pfd[1], "hello\n", 6) == -1)
        syserr("write");
    if (close(pfd[1]) == -1)
        syserr("close4");
}
```

Das war die nicht sonderlich überraschende Ausgabe:

```
hello
```

Achten Sie darauf, daß wir **dup** auf 0 geprüft haben anstatt auf –1. Für diesen **dup**-Aufruf ist es unmöglich, etwas anderes als 0 zu liefern — es sei denn, wir

haben einen Programmierfehler gemacht oder UNIX ist verrückt geworden. Deshalb haben wir uns nicht darum gekümmert, den Fall –1, bei dem wir **syserr** hätten aufrufen können, vom Fall, daß eine zu große bzw. falsche Nummer als Resultat geliefert wird, zu unterscheiden. Weiterhin haben wir den Filedeskriptor **pfd[0]** sofort nach Verdopplung geschlossen, da er dann nicht mehr gebraucht wird. Nach dem Schreiben schließt der Vater den Filedeskriptor **pfd[1]**, so daß **cat** *end-of-file* erhält.

Per Zufall hätten wir allerdings Probleme bekommen können: Wenn nämlich Filedeskriptor 0 beim Aufruf von **pipe** geschlossen gewesen wäre, so wäre dann eine der Nummern in **pfd** die Null gewesen. Beim Schließen von Filedeskriptor 0 hätten wir dann ein Ende der Pipe verloren und **dup** wäre schiefgegangen. Deshalb ist es lebenswichtig, die Standard-Filedeskriptoren (0, 1 und 2) immer aktiv zu lassen und sie auch an alle Söhne zu vererben. Zu viele Algorithmen, wie der, den wir gerade gezeigt haben, unterstellen eine Standard-Umgebung und laufen Amok, wenn sie keine finden.[3]

Wir müssen uns nicht darauf beschränken, nur vom Vater zum Sohn Daten über eine Pipe zu schicken. Das folgende Beispiel verhält sich äquivalent zur Shell-Kommandozeile

```
who | wc
```

deren Ausführung anzeigt, wie viele Benutzer gerade angemeldet sind.

```
who_wc()  /* who | wc */
{
    int pfd[2];

    if (pipe(pfd) == -1)
        syserr("pipe");
    switch (fork())
    {
        case -1:
            syserr("fork");
```

[3] Vor vielen Jahren zwang ich zufälligerweise einmal ein UNIX-System in die Knie, indem ich den Texteditor aufrief, als bereits zuviele Filedeskriptoren aktiv waren. Der Editor versuchte immer wieder vergebens seine temporäre Datei zu öffnen, und füllte dabei den Katalog **/tmp** mit allen Namen, die für temporäre Dateien möglich waren. Der Editor glaubte, daß diese Namen durch andere Editorprozesse belegt wären, aber tatsächlich war die Tatsache schuld, daß alle 15 (so viele waren es damals) Filedeskriptoren bereits aktiv waren (**errno** war damals noch nicht erfunden oder es wurde eben schlichtweg nicht darauf geprüft). Die Folge war, daß niemand editieren konnte und jedermann annahm, das System sei abgestürzt. Durch Aufräumen in **/tmp** wurde das System wieder in Ordnung gebracht.

```
        case 0:
            if (close(1) == -1)
                syserr("close");
            if (dup(pfd[1]) != 1)
                fatal("dup");
            if (close(pfd[0]) == -1 || close(pfd[1]) == -1)
                syserr("close2");
            execlp("who", "who", NULL);
            syserr("execl");
    }
    switch (fork())
    {
        case -1:
            syserr("fork");
        case 0:
            if (close(0) == -1)
                syserr("close3");
            if (dup(pfd[0]) != 0)
                fatal("dup2");
            if (close(pfd[0]) == -1 || close(pfd[1]) == -1)
                syserr("close4");
            execlp("wc", "wc", NULL);
            syserr("execl2");
    }
    if (close(pfd[0]) == -1 || close(pfd[1]) == -1)
        syserr("close5");
    while (wait(NULL) != -1)
        ;
}
```

Dies ist die Ausgabe:[4]

```
1 5 30
```

Natürlich ist es viel einfacher, für solche Kommandozeilen eine Shell zu benutzen als ein Programm dafür zu schreiben. Deshalb werden wir im nächsten Abschnitt eine solche Shell bauen.

[4] Ich benütze einen Personal-Computer, so bin immer nur ich angemeldet.

6.4 Eine echte Shell

Unsere Shell ist eine Untermenge der Standard-Shell **sh** in UNIX. Folgende Punkte kennzeichnen sie:

- Ein *einfaches Kommando* besteht aus einem Kommandonamen mit optional folgenden Argumenten, die durch Leer- oder Tabulatorzeichen voneinander getrennt sind. Jedes dieser Argumente ist ein einzelnes Wort oder eine mit doppelten Anführungszeichen (") umgebene Zeichenkette. Eine so zitierte Zeichenkette kann beliebige Sonderzeichen enthalten (|, Semikolon, &, >, <, Leer- oder Tabulatorzeichen und *Newline*). Ein eingeschlossenes doppeltes Anführungszeichen oder ein Rückwärtsschrägstrich muß mit einem führenden Rückwärtsschrägstrich zitiert werden (\" oder \\). Ein Kommando kann bis zu 20 Argumente haben, von denen jedes aus bis zu 200 Zeichen bestehen darf.

- Die Standard-Eingabe (Filedeskriptor 0) eines einfachen Kommandos kann auf eine Datei umgelenkt werden, indem man dem Dateinamen < voranstellt. Ähnlich kann auch die Standard-Ausgabe (Filedeskriptor 1) mittels > umgelenkt werden, wobei die Länge der Ausgabedatei erst einmal auf 0 verkürzt wird. Mit >> dagegen kann man die Ausgabe an eine Datei anhängen. Falls die Datei noch nicht existiert, so wird sie erzeugt.

- Eine *Pipeline* besteht aus einer Folge von einem oder mehreren einfachen Kommandos, die durch | voneinander getrennt sind. Alle einfachen Kommandos in einer Pipeline (außer dem letzten) sind über ihre Standard-Ausgabe mit der Standard-Eingabe des rechten Nachbars verbunden.

- Eine Pipeline wird durch *Newline*, Semikolon oder & beendet. In den ersten beiden Fällen wartet die Shell, bis das am weitesten rechts stehende einfache Kommando beendet ist, bevor sie weitermacht. Im Falle von & wartet die Shell nicht: Sie gibt für jedes einfache Kommando in der Pipeline die Prozeßnummer aus, und während der Ausführung dieser einfachen Kommandos werden *Interrupt*- und *Quit*-Signale ignoriert.

- Als eingebaute Kommandos gibt es Zuweisungen, **set**, **export** und **cd**. Die ersten drei sind in Abschnitt 5.3 beschrieben, und **cd** verhält sich wie gewohnt.

Der erste Schritt ist, die Eingabezeilen in Symbole, sogenannte *Token*, zu zerlegen. Ein Symbol ist eine Gruppe von Zeichen, die eine syntaktische Einheit bilden. Beispiele hierfür sind etwa Worte, zitierte Zeichenketten und spezielle Symbole wie & oder >>. Jedes Symbol wird durch eine symbolische Konstante wie folgt repräsentiert:

T_WORD	Ein Argument oder ein Dateiname. Falls zitiert, werden die Anführungszeichen nach Erkennen des Symbols entfernt.
T_BAR	Das Symbol |.
T_AMP	Das Symbol &.
T_SEMI	Das Symbol ;.
T_GT	Das Symbol >.
T_GTGT	Das Symbol >>.
T_LT	Das Symbol <.
T_NL	Das *Newline*-Zeichen.
T_EOF	Ein spezielles Symbol, durch das das Dateiende angezeigt wird. Falls die Standard-Eingabe ein Terminal ist, so hat der Benutzer *EOT* (*Control*-D) getippt.

Die Arbeit der lexikalischen Analyse besteht darin, die Eingabe zu lesen und die Eingabezeichen in entsprechende Symbole umzuwandeln. Falls das Symbol identisch zu **T_WORD** ist, so wird außerdem eine Zeichenkette zurückgegeben, die gerade die Zeichen enthält, die **T_WORD** in Wirklichkeit darstellt. Bei den anderen Symbolen sind die tatsächlich gelesenen Zeichen offensichtlich. Die Funktion zur lexikalischen Analyse sollte unwichtige Zeichen, wie Leerzeichen zwischen Argumenten, überlesen, ohne irgendetwas zurückzugeben.[5]

Unsere Funktion zur lexikalischen Analyse ist eine endliche Maschine: Werden Zeichen gelesen, so werden sie entweder sofort als Symbol erkannt oder sie werden gesammelt (z.B. bei Buchstaben eines Worts). Mit jedem Zeichen kann die Funktion zur lexikalischen Analyse in einen neuen Zustand überwechseln, der dazu dient, die Interpretation weiterer Zeichen zu steuern. Wenn zum Beispiel eine zitierte Zeichenkette gelesen wird, so werden Leerzeichen anders behandelt als außerhalb von Anführungszeichen. Für unsere Shell benötigen wir vier Zustände:

NEUTRAL	Der Startzustand: Leer- und Tabulatorzeichen werden überlesen; |, &, Semikolon, < und *Newline* werden sofort als Symbole erkannt. Das Zeichen > bewirkt einen Wechsel in den Zustand **GTGT**, der noch auf das nächste Zeichen wartet, da > und >> verschiedene Symbole sind. Ein " veranlaßt einen Wechsel in den Zustand **INQUOTE**, der eine zitierte Zeichenkette sammelt.

[5] Viele UNIX-Systeme beinhalten ein Kommando namens **lex**, das aufgrund einer Beschreibung der zu erkennenden Symbole automatisch eine Funktion zur lexikalischen Analyse erzeugen kann. Da **lex** zum einen kompliziert in der Benutzung ist, und zum anderen die entstehenden Funktionen zur lexikalischen Analyse groß und langsam sind, ist es in der Regel besser, solche von Hand zu schreiben. Es sind nicht wirklich schwierige Programme.

Alles andere wird als Beginn eines nichtzitierten Worts interpretiert, wobei das Zeichen in einem Puffer gespeichert wird und der neue Zustand **INWORD** ist.

GTGT Dieser Zustand bedeutet, daß gerade eben > gelesen wurde. Falls das nächste Zeichen wieder > ist, so wird das Symbol **T_GTGT** zurückgegeben, andernfalls das Symbol **T_GT**. Da wir im zweiten Fall aber ein Zeichen zuviel gelesen haben, müssen wir es in die Eingabe zurückstellen (für diesen Zweck gibt es in der Standard-Bücherei der Ein- und Ausgabe die Funktion **ungetc**).

INQUOTE Dieser Zustand bedeutet, daß gerade eben ein doppeltes Anführungszeichen gelesen wurde. Nun werden die weiteren Zeichen in einem Puffer gesammelt, bis das abschließende doppelte Anführungszeichen gelesen wird. Dann werden das Symbol **T_WORD** und die gespeicherte Zeichenkette zurückgegeben. Bei Auftreten des Fluchtsymbols \ müssen besondere Maßnahmen ergriffen werden.

INWORD Dieser Zustand bedeutet, daß das erste Zeichen eines Worts gelesen und in den Puffer gesteckt wurde. Weitere Zeichen werden solange gesammelt, bis ein Zeichen gelesen wird, das nicht zu einem Wort gehört (z.B. |). Dann wird dieses unerwünschte Zeichen in die Eingabe zurückgestellt und das Symbol **T_WORD** zurückgeliefert.

Mit diesen Erklärungen sollte der Programmtext von **gettoken**, unserer Funktion zur lexikalischen Analyse, leicht verständlich sein:

```
typedef enum {T_WORD, T_BAR, T_AMP, T_SEMI, T_GT,
              T_GTGT, T_LT,  T_NL,  T_EOF}
        TOKEN;

static TOKEN gettoken(word)  /* Sammeln und Symbol erkennen */
    char * word;
{
    enum {NEUTRAL, GTGT, INQUOTE, INWORD} state = NEUTRAL;
    int c;
    char * w;
```

```
w = word;
while ((c = getchar()) != EOF)
{
    switch (state)
    {
        case NEUTRAL:
            switch (c)
            {
                case ';':
                    return(T_SEMI);
                case '&':
                    return(T_AMP);
                case '|':
                    return(T_BAR);
                case '<':
                    return(T_LT);
                case '\n':
                    return(T_NL);
                case ' ':
                case '\t':
                    continue;
                case '>':
                    state = GTGT;
                    continue;
                case '"':
                    state = INQUOTE;
                    continue;
                default:
                    state = INWORD;
                    * w++ = c;
                    continue;
            }
        case GTGT:
            if (c == '>')
                return(T_GTGT);
            ungetc(c, stdin);
            return(T_GT);
```

```
            case INQUOTE:
                switch (c)
                {
                    case '\\':
                        * w++ = getchar();
                        continue;
                    case '"':
                        * w = '\0';
                        return(T_WORD);
                    default:
                        * w++ = c;
                        continue;
                }
            case INWORD:
                switch (c)
                {
                    case ';':
                    case '&':
                    case '|':
                    case '<':
                    case '>':
                    case '\n':
                    case ' ':
                    case '\t':
                        ungetc(c, stdin);
                        * w = '\0';
                        return(T_WORD);
                    default:
                        * w++ = c;
                        continue;
                }
        }
    }
    return(T_EOF);
}
```

Wenn man ein großes Programm schreibt, ist es üblich, dieses in kleinen Stücken zu testen. Dies gibt nicht nur ein frühes Feedback (über die Funktionsweise des Programms), es macht auch die Fehlersuche leichter, da einfach weniger Programmtext zu durchsuchen ist. Hier ist ein Programm, das wir zum Testen von **gettoken** geschrieben haben (und das man nach dem Testen wegwerfen kann):

```
main()  /* gettoken() testen */
{
    char word[200];

    while (1)
        switch (gettoken(word))
        {
            case T_WORD:
                printf("T_WORD <%s>\n", word);
                break;
            case T_BAR:
                printf("T_BAR\n");
                break;
            case T_AMP:
                printf("T_AMP\n");
                break;
            case T_SEMI:
                printf("T_SEMI\n");
                break;
            case T_GT:
                printf("T_GT\n");
                break;
            case T_GTGT:
                printf("T_GTGT\n");
                break;
            case T_LT:
                printf("T_LT\n");
                break;
            case T_NL:
                printf("T_NL\n");
                break;
            case T_EOF:
                printf("T_EOF\n");
                exit(0);
        }
}
```

Als wir das Programm laufen ließen und folgende Zeile tippten (gefolgt von *EOT*):

```
sort <inf | pr -h "Sept. Results" >>outf&
```

erhielten wir folgende Ausgabe:

```
T_WORD <sort>
T_LT
T_WORD <inf>
T_BAR
T_WORD <pr>
T_WORD <-h>
T_WORD <Sept. Results>
T_GTGT
T_WORD <outf>
T_AMP
T_NL
T_EOF
```

Dieser Test beweist nicht die Richtigkeit von **gettoken**, aber er ermutigt uns doch genug, um mit der Shell weiterzumachen.

Der nächste Schritt ist, eine Funktion zu schreiben, die ein einfaches Kommando bearbeiten kann, das durch |, &, Semikolon oder *Newline* beendet ist. Diese Funktion nennen wir **command**. Argumente werden einfach in einem Vektor namens **argv** zur späteren Benutzung in einem Aufruf von **execvp** hinterlegt. Für die Standard-Eingabe gibt es drei Möglichkeiten: die Voreinstellung (Filedeskriptor 0), eine Datei (falls < vorkam) oder das lesbare Ende einer Pipe (falls *vor* dem einfachen Kommando | stand). Für die Standard-Ausgabe gibt es vier Möglichkeiten: die Voreinstellung (Filedeskriptor 1), eine zu erzeugende oder zu verkürzende Datei (falls > vorkam), eine zu erzeugende oder zu verlängernde Datei (falls >> vorkam) oder das schreibbare Ende einer Pipe (falls *nach* dem einfachen Kommando | stand). Das Symbol, das von **gettoken** geliefert wird, erzählt uns, welche der Möglichkeiten tatsächlich vorliegt.

Während **command** die Symbole verarbeitet, werden folgende Variablen zur Speicherung der aktuellen Situation der Standard-Eingabe und der Standard-Ausgabe für ein einfaches Kommando benützt:

srcfd	Der Filedeskriptor der Quelle, zu Anfang 0. Falls die Eingabe mittels < umgelenkt wird, so wird **srcfd** auf **BADFD** (–2) gesetzt und **srcfile** enthält dann den Dateinamen. Kommt die Eingabe dagegen aus einer Pipe, so wird **srcfd** auf eine Zahl größer als 0 gesetzt.
srcfile	Die Quelldatei. Diese Variable ist nur dann sinnvoll belegt, wenn das einfache Kommando < enthält.
dstfd	Der Filedeskriptor des Ziels, zu Anfang 1. Falls die Ausgabe mittels > oder >> umgelenkt wird, so wird **dstfd** auf **BADFD** gesetzt und **dstfile** enthält dann den Dateinamen. Falls die Ausgabe eine Pipe ist, so enthält **dstfd** eine Zahl größer als 1.

dstfile Die Zieldatei. Diese Variable ist nur dann sinnvoll belegt, wenn das einfache Kommando > oder >> enthält.

append Diese Boole'sche Variable wird bei Umlenkung der Ausgabe mittels >> auf **TRUE**, bei Umlenkung mittels > auf **FALSE** gesetzt. Anderenfalls ist sie nicht initialisiert.

makepipe Dies ist ein Argument für **command**. Falls **makepipe** auf **TRUE** gesetzt ist, so verlangt der Aufrufer, daß **command** eine Pipe erzeugt, deren lesbares Ende als Standard-Eingabe benützt und den Filedeskriptor des schreibbaren Endes an den Aufrufer zurückgibt, der diesen dann für seine eigenen Zwecke benützen kann. Dieses Vorgehen wird in Kürze erklärt.

Während **command** die Symbole erkennt, prüft die Funktion gleichzeitig auf logische Fehler: mehrfaches Vorkommen von < oder >, Auftreten von >, obwohl das einfache Kommando mit | endet, Auftreten von <, obwohl vor dem einfachen Kommando | steht, und so weiter. Es ist ganz interessant, daß die Standard-Shell unter UNIX solche Anomalien nicht prüft: Tatsächlich kann man eine Kommandozeile wie die folgende ausführen:

```
who > outf | wc
```

Die Funktion **command** gibt das Symbol zurück, das die Pipeline beendet hat, da der Typ dieses Ende-Symbols die weitere Verarbeitung beeinflußt: Ist & das Ende-Symbol der Pipeline, so wird auf das Ende des am weitesten rechts stehenden einfachen Kommandos nicht gewartet; weiterhin werden bei der Ausführung aller einfachen Kommandos in der Pipeline *Interrupt-* und *Quit*-Signale ignoriert. Wenn auf das am weitesten rechts stehende einfache Kommando gewartet wird, so muß dessen Prozeßnummer über das Argument **waitpid** zurückgeliefert werden. Stand am Ende *Newline*, so wird der Benutzer zu einem neuen Kommando aufgefordert.

Der raffinierteste Aspekt bei **command** ist der rekursive Aufruf, falls | einem einfachen Kommando folgt. Rekursive Aufrufe fahren solange fort, bis eins der anderen Ende-Zeichen (Semikolon, & oder *Newline*) erreicht wird. Jeder rekursive Aufruf ist für die Erzeugung einer Pipe verantwortlich (mit Hilfe des Systemaufrufs **pipe**); deshalb ist sein Argument **makepipe** auch auf **TRUE** gesetzt. Der Filedeskriptor des schreibbaren Endes der Pipe wird zurückgegeben (über ein anderes Argument namens **pipefd**). Ein einfaches Kommando wird erst nach Rückkehr aus der Rekursion aufgerufen (durch die Funktion **invoke**), da hierzu das schreibbare Ende der Pipe nötig ist. Außerdem muß auch, wie oben schon bemerkt, das Ende-Symbol der Pipeline bekannt sein, bevor eines der dazugehörigen einfachen Kommandos ausgeführt werden kann, da **invoke** etwas über die Behandlung von Signalen wissen will. Deshalb werden Pipelines von links nach rechts abgearbeitet, die einfachen Kommandos aber von rechts nach links. Für das am weitesten links stehende einfache Kommando gibt es einen Aufruf von **command**, wobei **makepipe** den Wert

FALSE besitzt. Für jedes weitere einfache Kommando existiert ein weiterer rekursiver Aufruf, wobei **makepipe** diesmal den Wert **TRUE** besitzt. So wie sich der Stack der Aufrufe von **command** abbaut, werden die einfachen Kommandos aufgerufen — alle nötige Information ist dann vorhanden.

Hier ist der Text von **command**. Er ist ein sorgfältiges Studium wert.

```
                                /* einfaches Kommando ausfuehren */
static TOKEN command(waitpid, makepipe, pipefdp)
    int * waitpid, * pipefdp;
    BOOLEAN makepipe;
{
    TOKEN token, term, gettoken();
    int argc, srcfd, dstfd, pid, pfd[2];
    char * argv[MAXARG + 1], srcfile[MAXFNAME], dstfile[MAXFNAME];
    char word[MAXWORD], * malloc();
    BOOLEAN append;

    argc = 0;
    srcfd = 0;
    dstfd = 1;
    while (1)
    {
        switch (token = gettoken(word))
        {
            case T_WORD:
                if (argc == MAXARG)
                {
                    fprintf(stderr, "Too many args\n");
                    break;
                }
                if ((argv[argc] = malloc(strlen(word) + 1)) == NULL)
                {
                    fprintf(stderr, "Out of arg memory\n");
                    break;
                }
                strcpy(argv[argc], word);
                argc++;
                continue;
```

```
case T_LT:
    if (makepipe)
    {
        fprintf(stderr, "Extra <\n");
        break;
    }
    if (gettoken(srcfile) != T_WORD)
    {
        fprintf(stderr, "Illegal <\n");
        break;
    }
    srcfd = BADFD;
    continue;
case T_GT:
case T_GTGT:
    if (dstfd != 1)
    {
        fprintf(stderr, "Extra > or >>\n");
        break;
    }
    if (gettoken(dstfile) != T_WORD)
    {
        fprintf(stderr, "Illegal > or >>\n");
        break;
    }
    dstfd = BADFD;
    append = token == T_GTGT;
    continue;
case T_BAR:
case T_AMP:
case T_SEMI:
case T_NL:
    argv[argc] = NULL;
    if (token == T_BAR)
    {
        if (dstfd != 1)
        {
            fprintf(stderr, "> or >> conflicts with |\n");
            break;
        }
        term = command(waitpid, TRUE, &dstfd);
    }
```

```
                else
                    term = token;
                if (makepipe)
                {
                    if (pipe(pfd) == -1)
                        syserr("pipe");
                    * pipefdp = pfd[1];
                    srcfd = pfd[0];
                }
                pid = invoke(argc, argv, srcfd, srcfile, dstfd,
                             dstfile, append, term == T_AMP);
                if (token != T_BAR)
                    * waitpid = pid;
                if (argc == 0 && (token != T_NL || srcfd > 1))
                    fprintf(stderr, "Missing command\n");
                while (--argc >= 0)
                    free(argv[argc]);
                return(term);
            case T_EOF:
                exit(0);
        }
    }
}
```

Wenn **command** aufgrund des Symbols **T_BAR** die Notwendigkeit einer Pipe erkannt hat, stellt sich die folgende Frage: Warum wird das Erzeugen der Pipe auf den nächsten Aufruf von **command** verschoben, anstatt diese selbst zu kreieren und nur den Filedeskriptor des lesbaren Endes weiterzureichen? Das macht man, um Filedeskriptoren zu sparen. Wenn der Systemaufruf **pipe** vor (anstatt nach) **command** aufgerufen würde, würde jeder Grad der Rekursion — also jedes einfache Kommando — zwei Filedeskriptoren für die Pipe aufbrauchen, die erst nach Rückkehr des rekursiven Aufrufs geschlossen werden könnten. Da es nur 20 Filedeskriptoren gibt, wären Pipelines auf maximal acht einfache Kommandos beschränkt. Dadurch, daß der Leser die Pipe erzeugen muß, können wir **pipe** unmittelbar vor **invoke** aufrufen, falls dies der Aufrufer von uns verlangt — so erlauben wir Pipelines beliebiger Länge.

Hier ist nun das Hauptprogramm, das das erste Mal **command** aufruft. Es ist an das Hauptprogramm der Einweg-Shell aus Abschnitt 5.3 angelehnt.

```
main()  /* wirkliche Shell */
{
    char * prompt;
    int pid, fd;
    TOKEN term, command();

    ignoresig();
    if (!EVinit())
        fatal("can't initialize environment");
    if ((prompt = EVget("PS2")) == NULL)
        prompt = "> ";
    printf("%s", prompt);
    while (1)
    {
        term = command(&pid, FALSE, NULL);
        if (term != T_AMP && pid != 0)
            waitfor(pid);
        if (term == T_NL)
            printf("%s", prompt);
        for (fd = 3; fd < 20; fd++)
            (void)close(fd);  /* Fehler ignorieren */
    }
}
```

Der Aufruf von **ignoresig** bewirkt, daß *Interrupt*- und *Quit*-Signale ignoriert werden — wir wollen unsere Shell ja nicht umbringen, wenn wir die *Interrupt*- oder die *Quit*-Taste drücken. Den Programmtext für **ignoresig** zeigen wir in Abschnitt 8.3. Das Ende-Symbol der Pipeline, das von **command** geliefert wird, bestimmt, ob wir auf das Ende des am weitesten rechts stehenden einfachen Kommandos warten müssen (wir werden **waitfor** in Kürze sehen), und ob wir sofort zur erneuten Eingabe auffordern müssen. Nach jeder Pipeline schließen wir alle bis auf die Standard-Filedeskriptoren, für den Fall daß ein Fehler **invoke** dazu gebracht hat, einen Filedeskriptor offenzulassen.

command ruft **invoke** zur Ausführung eines einfachen Kommandos auf. Hierbei werden die Kommando-Argumente (**argc** und **argv**) und die bereits früher beschriebenen Variablen, die die Quelle und das Ziel beschreiben (**srcfd**, **srcfile**, **dstfd**, **dstfile** und **append**) übergeben. Das letzte Argument bestimmt, ob **invoke** das einfache Kommando im Hintergrund ausführen soll. Das Schema von **fork** und **exec**, das hierbei benützt wird, sollte mittlerweile vertraut sein (falls nicht, lesen Sie doch noch einmal Abschnitt 5.4):

```
                                        /* einfaches Kommando ausfuehren */
static int invoke(argc, argv, srcfd, srcfile, dstfd,
                  dstfile, append, bckgrnd)
    int argc, srcfd, dstfd;
    char * argv[], * srcfile, * dstfile;
    BOOLEAN append, bckgrnd;
{
    int pid;
    BOOLEAN builtin();

    if (argc == 0 || builtin(argc, argv, srcfd, dstfd))
        return(0);
    switch (pid = fork())
    {
        case -1:
            fprintf(stderr, "Can't create new process\n");
            return(0);
        case 0:
            if (!bckgrnd)
                entrysig();
            if (!EVupdate())
                fatal("can't update environment");
            redirect(srcfd, srcfile, dstfd, dstfile, append, bckgrnd);
            execvp(argv[0], argv);
            fprintf(stderr, "Can't execute %s\n", argv[0]);
            exit(0);
        default:
            if (srcfd > 0 && close(srcfd) == -1)
                syserr("close src");
            if (dstfd > 1 && close(dstfd) == -1)
                syserr("close dst");
            if (bckgrnd)
                printf("%d\n", pid);
            return(pid);
    }
}
```

Das Kommando, das ausgeführt werden soll, kann eingebaut sein. Falls dem so ist, liefert **builtin**, das wir in Kürze sehen, den Wert **TRUE**. Falls nicht, wird ein Sohn-Prozeß erzeugt, um das einfache Kommando auszuführen. Erinnern Sie sich, daß *Interrupt*- und *Quit*-Signale bereits ignoriert werden. Falls das Kommando nicht im Hintergrund laufen soll, werden die Reaktionen auf Signale mittels eines

Aufrufs der Funktion **entrysig** (Einzelheiten finden Sie in Abschnitt 8.3) auf ihren ursprünglichen Zustand (vor Eintritt in die Shell) gesetzt.

invoke ruft **redirect** auf, um die Ein- und Ausgabe umzulenken und um sicherzugehen, daß Quelle und Ziel die Filedeskriptoren 0 und 1 darstellen. Hier ist der Programmtext von **redirect**:

```
                                        /* Umlenkung der Ein- und Ausgabe */
static void redirect(srcfd, srcfile, dstfd, dstfile, append, bckgrnd)
    int srcfd, dstfd;
    char * srcfile, * dstfile;
    BOOLEAN append, bckgrnd;
{
    int flags, fd;
    long lseek();

    if (srcfd == 0 && bckgrnd)
    {
        strcpy(srcfile, "/dev/null");
        srcfd = BADFD;
    }
    if (srcfd != 0)
    {
        if (close(0) == -1)
            syserr("close");
        if (srcfd > 0)
        {
            if (dup(srcfd) != 0)
                fatal("dup");
        }
        else if (open(srcfile, O_RDONLY, 0) == -1)
        {
            fprintf(stderr, "Can't open %s\n", srcfile);
            exit(0);
        }
    }
    if (dstfd != 1)
    {
        if (close(1) == -1)
            syserr("close");
```

```
        if (dstfd > 1)
        {
            if (dup(dstfd) != 1)
                fatal("dup");
        }
        else
        {
            flags = O_WRONLY | O_CREAT;
            if (!append)
                flags |= O_TRUNC;
            if (open(dstfile, flags, 0666) == -1)
            {
                fprintf(stderr, "Can't create %s\n", dstfile);
                exit(0);
            }
            if (append)
                if (lseek(1, 0L, 2) == -1)
                    syserr("lseek");
        }
    }
    for (fd = 3; fd < 20; fd++)
        (void)close(fd);  /* Fehler ignorieren */
}
```

Wenn ein Hintergrund-Kommando seine Standard-Eingabe nicht in eine Datei oder in eine Pipe umgelenkt hat, so ergreifen wir die Vorsichtsmaßnahme, sie in die Gerätedatei **/dev/null** umzulenken, was beim Lesen sofort *end-of-file* herbeiführt. Dies vermeidet die unschöne Situation, daß sich zwei Prozesse um die Tastatureingaben des Benutzers streiten. Der Rest von **redirect** sollte klar sein. Denken Sie daran, daß **open** so wie **dup** immer den Filedeskriptor mit der kleinsten verfügbaren Nummer wählt.

Im ganzen Programmtext, den der Sohn-Prozeß ausführt, rufen wir bei einem Fehler **exit(0)** statt des gebräuchlicheren **exit(1)** auf; dies dient zur Unterscheidung von Fehlern, die die Shell direkt erfährt (zum Beispiel, daß eine Datei nicht geöffnet werden kann), gegenüber Fehlern, die in dem aufgerufenen Programm entstehen. Im zweiten Fall wollen wir, daß die Shell den Exit-Status ausgibt, während uns im ersten Fall eine einfache Fehlermeldung genügt. Tatsächlich wäre es sogar verwirrend, einen Exit-Status auszugeben, wenn ein Sohn-Prozeß zu Ende geht, ohne den **exec**-Aufruf beendet zu haben — 'exit' woraus? Nur UNIX-Systemprogrammierern wird zugetraut, über die Trennung von **fork** und **exec** Bescheid zu wissen.[6]

[6] Dies ist keine A.d.Ü. A.d.Ü.

waitfor (aufgerufen im Hauptprogramm) ist eine Erweiterung eines Programms aus Abschnitt 5.6 (die Funktion **execut3**). Die Funktion **waitfor** erfährt, auf welchen Prozeß sie zu warten hat; während dieser Zeit erfährt sie aber auch von der Termination anderer Prozesse. Falls nötig, benützt **waitfor** die Funktion **statusprt**, um die Prozeßnummer und den Grund des Prozeßendes auszugeben, falls dieses abnormal war. Wenn der gesuchte Prozeß zu Ende geht, ruft **waitfor** die Funktion **statusprt** auf und kehrt zurück. Hier ist der Programmtext:

```
static void waitfor(pid)  /* auf Sohn-Prozess warten */
    int pid;
{
    int wpid, status;

    while ((wpid = wait(&status)) != pid && wpid != -1)
        statusprt(wpid, status);
    if (wpid == pid)
        statusprt(0, status);
}
```

Schließlich kommen wir noch zur Funktion **builtin**. Der größte Teil davon stammt aus Abschnitt 5.3; hinzugefügt haben wir die Behandlung des **cd**-Kommandos. **cd** ohne Argumente wechselt in den Katalog, der durch die Environment-Variable **HOME** dargestellt wird.

```
                                /* eingebaute Kommandos ausfuehren */
static BOOLEAN builtin(argc, argv, srcfd, dstfd)
    int argc, srcfd, dstfd;
    char * argv[];
{
    char * path;

    if (strchr(argv[0], '=') != NULL)
        asg(argc, argv);
    else if (strcmp(argv[0], "export") == 0)
        export(argc, argv);
    else if (strcmp(argv[0], "set") == 0)
        set(argc, argv);
    else if (strcmp(argv[0], "cd") == 0)
    {
        if (argc > 1)
            path = argv[1];
```

```
        else
            if ((path = EVget("HOME")) == NULL)
                path = ".";
        if (chdir(path) == -1)
            fprintf(stderr, "%s: bad directory\n", path);
    }
    else
        return(FALSE);
    if (srcfd != 0 || dstfd != 1)
        fprintf(stderr, "Illegal redirection or pipeline\n");
    return(TRUE);
}
```

Vielleicht haben Sie bemerkt, daß manche Fehler, die unsere Shell entdeckt, sich tödlich auswirken (Aufrufe von **syserr** und **fatal**), während andere gerade die Ausgabe einer Fehlermeldung bewirken, und die Shell danach weitermacht. Wir haben uns entschieden, daß die "unmöglichen" Fehler terminierend wirken, da ihr Auftreten eine tödliche Wunde im Betriebssystem anzeigt. Dies ist natürlich ein Kompromiß: Einerseits wollen wir diese Fehler nicht ignorieren, da wir wissen, daß auch Unmögliches geschieht; andererseits sind wir zu faul, in einer Situation, die wahrscheinlich doch nie eintritt, eine Fehlermeldung auszugeben und sinnvoll zurückzukehren. Manchmal täuschen wir uns — ein tödlicher Fehler tritt auf und ist eben doch nicht unmöglich. Dann müssen wir den Programmtext ändern, so daß dieser Fehler separat behandelt wird.

Wir werden kein Beispiel einer Sitzung mit dieser Shell zeigen, da dies überflüssig ist — sie verhält sich wie die Standard-Shell.[7]

6.5 Bidirektionale Pipes

Nun wollen wir uns von den Ein-Weg-Pipelines, wie sie die Shell benützt, hinweg zu sogenannten Zwei-Weg-Pipelines bewegen. Die Standard-Shell **sh** bietet keine Notation, um eine Zwei-Weg-Kommunikation zwischen zwei Prozessen aufzubauen (es gibt allerdings experimentelle Shells, die dies können). Zwei-Weg-Pipelines werden über ein C-Programm aufgebaut.

Wir wollen mit einem ziemlich einfachen Beispiel beginnen. Innerhalb eines Programms wollen wir **sort** ausführen, um Daten zu sortieren. Natürlich könnten wir dies wie folgt tun:

```
system("sort <data >output");
```

[7] Nicht ganz. A.d.Ü.

Dann könnten wir den Inhalt von **output** lesen, um auf die sortierten Daten zuzugreifen. So wollen wir es aber gerade nicht machen — wir wollen die Daten über eine Pipe zu **sort** lenken, und **sort** soll die sortierte Ausgabe über eine Pipe an uns zurückschicken. Da **sort** seine Standard-Eingabe liest und seine Standard-Ausgabe schreibt (**sort** ist ein sogenannter *Filter*), sollten wir **sort** in der von uns gewünschten Weise benutzen können.[8] Wir wissen bereits, wie wir einen beliebigen Filedeskriptor zwingen können, die Standard-Ein- oder Ausgabe eines Prozesses zu sein.

So wie kleine Kinder vor dem Laufen erst das Fallen lernen, so wollen auch wir diese Aufgabe erst einmal nicht ganz richtig lösen. So lernen wir mehr darüber, wie man sie richtig löst, als wenn wir die richtige Lösung auf Anhieb zeigen.

Da jede Pipe sowohl ein lesbares als auch ein schreibbares Ende besitzt, und da diese beiden Filedeskriptoren an einen Sohn-Prozeß vererbt werden, kommen wir mit einer Pipe aus. **sort** kann aus dieser Pipe lesen, um seine Eingabe zu bekommen und danach in die Pipe schreiben, um die sortierten Daten zurückzuschicken. Der Vater-Prozeß besitzt auch Filedeskriptoren für die Pipe. Er schreibt die unsortierten Daten in die Pipe und liest die sortierten aus dieser heraus. Es folgt nun ein Programm, das aus der Datei **data** Daten liest und **sort** ausführt, um diese zu sortieren. Die sortierten Daten werden dann ausgegeben.

```
fsort0()  /* Datei sortieren (falsch) */
{
    int pfd[2], fd, nread;
    char buf[512];

    if (pipe(pfd) == -1)
        syserr("pipe");
    switch (fork())
    {
        case -1:
            syserr("fork");
        case 0:
            if (close(0) == -1)
                syserr("close");
            if (dup(pfd[0]) != 0)
                fatal("dup");
            if (close(1) == -1)
                syserr("close");
            if (dup(pfd[1]) != 1)
                fatal("dup");
```

[8] **sort** ist ein Pseudo-Filter, der zur Zwischenspeicherung temporäre Dateien verwendet. A.d.Ü.

```
            if (close(pfd[0]) == -1 || close(pfd[1]) == -1)
                syserr("close");
            execlp("sort", "sort", NULL);
            syserr("execl");
        }
        if ((fd = open("data", 0)) == -1)
            syserr("open");
        while ((nread = read(fd, buf, sizeof(buf))) != 0)
        {
            if (nread == -1)
                syserr("read");
            if (write(pfd[1], buf, nread) == -1)
                syserr("write");
        }
        if (close(fd) == -1 || close(pfd[1]) == -1)
            syserr("close");
        while ((nread = read(pfd[0], buf, sizeof(buf))) != 0)
        {
            if (nread == -1)
                syserr("read");
            if (write(1, buf, nread) == -1)
                syserr("write");
        }
        if (close(pfd[0]) == -1)
            syserr("close");
}
```

Das ist der Inhalt der Datei **data**:

```
peach
apple
orange
strawberry
plum
pear
cherry
banana
apricot
tomato
pineapple
mango
```

Als wir dieses Programm ausführten, gab es die Früchte in der gleichen Reihenfolge wie in der Datei **data** aus — und verklemmte sich dann. Wir mußten die *Interrupt*-Taste drücken, um es zu beenden. Was lief schief?

Zwei Probleme ergeben sich, wenn man nur eine Pipe benützt. Erstens: Nachdem der Vater-Prozeß die unsortierten Daten in die Pipe geschrieben hatte, begann er sofort mit dem Lesen der Pipe, in der Annahme, daß er die sortierte Ausgabe von **sort** lesen würde. Aber er kam **sort** zuvor und las so wieder seine eigene Ausgabe. Deshalb wurden die Daten in unsortierter Form ausgegeben. Dies erinnert an das erste Beispiel in Abschnitt 6.2.

Das zweite Problem verursachte die Deadlock-Situation. Der Sohn-Prozeß, der **sort** ausführt, begann seine Standard-Eingabe zu lesen, die aber unglücklicherweise bereits vom Vater-Prozeß geleert worden war. Leer oder nicht, **sort** würde in einem **read**-Systemaufruf blockiert werden, bis *end-of-file* auftritt, was nur bei Schließen des schreibbaren Endes passieren würde. Klar, der Vater-Prozeß hat dieses Ende bereits geschlossen, aber der Sohn-Prozeß hat es noch offen — schließlich soll er ja seine Ausgabe dorthin schreiben. Deshalb kam der Sohn-Prozeß nicht weiter. Generell läßt sich sagen, daß jeder Filter, der die gleiche Pipe liest und schreibt, eine Deadlock-Situation erzeugen wird.

Man könnte versucht sein, das erste Problem zu lösen, indem man den Vater-Prozeß, bevor dieser die Pipe liest, auf das Ende des Sohn-Prozesses warten läßt. Im ersten Moment klingt dies ganz gut, es funktioniert jedoch nicht: Wenn die Ausgabe des Sohn-Prozesses die Pipe füllt, was bei Sortieren einer größeren Datenmenge durchaus passieren kann, ist der Sohn im **write**-Aufruf blockiert. Da auch der Vater in **wait** blockiert ist, hätten wir wieder eine Deadlock-Situation.

Als nächstes könnte man zur Synchronisation einige Semaphore hinzufügen, um die Deadlock-Situation zu vermeiden. Allerdings hat diese Lösung zwei Nachteile. Erstens bietet keine UNIX-Version von AT&T vor System V Semaphore, so daß diese Lösung einem großen Teil der UNIX-Bevölkerung unzugänglich wäre. Zweitens, und das ist wichtiger, löst sich das Problem in Wohlgefallen auf, wenn man einfach *zwei* Pipes benutzt: Jede regelt dann den Datenverkehr in eine Richtung. Eine ist für den Datenfluß zu **sort**, und die andere für den Datenrückfluß verantwortlich. Hier nun die überarbeitete Fassung von **fsort0**, die korrekt funktioniert:

```
fsort()  /* Datei sortieren (richtig) */
{
    int pfdout[2], pfdin[2], fd, nread;
    char buf[512];

    if (pipe(pfdout) == -1 || pipe(pfdin) == -1)
        syserr("pipe");
```

```
    switch (fork())
    {
        case -1:
            syserr("fork");
        case 0:
            if (close(0) == -1)
                syserr("close");
            if (dup(pfdout[0]) != 0)
                fatal("dup");
            if (close(1) == -1)
                syserr("close");
            if (dup(pfdin[1]) != 1)
                fatal("dup");
            if (close(pfdout[0]) == -1 || close(pfdout[1]) == -1 ||
              close(pfdin[0]) == -1 || close(pfdin[1]) == -1)
                syserr("close");
            execlp("sort", "sort", NULL);
            syserr("execl");
    }
    if (close(pfdout[0]) == -1 || close(pfdin[1]) == -1)
        syserr("close");
    if ((fd = open("data", 0)) == -1)
        syserr("open");
    while ((nread = read(fd, buf, sizeof(buf))) != 0)
    {
        if (nread == -1)
            syserr("read");
        if (write(pfdout[1], buf, nread) == -1)
            syserr("write");
    }
    if (close(fd) == -1 || close(pfdout[1]) == -1)
        syserr("close");
    while ((nread = read(pfdin[0], buf, sizeof(buf))) != 0)
    {
        if (nread == -1)
            syserr("read");
        if (write(1, buf, nread) == -1)
            syserr("write");
    }
    if (close(pfdin[0]) == -1)
        syserr("close");
}
```

Nun die Ausgabe, die wir diesmal bekamen:

```
apple
apricot
banana
cherry
mango
orange
peach
pear
pineapple
plum
strawberry
tomato
```

Abgesehen davon, daß die Ausgabe sortiert ist, ist das Programm auch von allein zu Ende gegangen! Das Interessante an der richtigen Lösung ist, daß sie auch nicht mehr Filedeskriptoren benötigt, obwohl sie eine Pipe mehr benützt. In beiden Versionen müssen Vater und Sohn untereinander lesen und schreiben können, so daß jeder zwei Filedeskriptoren für eine Pipe benötigt. In der zweiten Version kann der Vater-Prozeß das lesbare Ende von **pfdout** und das schreibbare Ende von **pfdin** schließen.

Im allgemeinen ist eine Deadlock-Situation bei zwei Pipes immer noch möglich, wenn auch nicht in unserem Beispiel. Eine Deadlock-Situation würde auftreten, wenn der Vater-Prozeß nichts mehr in seine Ausgabe-Pipe schreibt, da diese gefüllt ist, und der Sohn-Prozeß, anstatt diese zu leeren, genügend Ausgabe zurück zum Vater-Prozeß schickt, so daß die andere Pipe auch blockiert wäre. Jeder Fall muß sorgfältig untersucht werden, um sicherzustellen, daß eine Deadlock-Situation nie auftreten kann, nicht nur bei kleinen Test-Datenmengen.

Es gibt aber auch andere Möglichkeiten, eine Deadlock-Situation zu verursachen. Um eine von diesen zu erforschen, wollen wir ein komplexeres Beispiel einer Zwei-Weg-Prozeßkommunikation ansehen. In diesem Beispiel ist der Sohn-Prozeß der Standard-Zeileneditor (**ed**), und der Vater-Prozeß benützt diesen als Unterprogramm[9], indem er dem Editor Kommandozeilen schickt und die entsprechende Ausgabe zurückerhält. Solch eine Anordnung könnte von einem Bildschirmeditor benutzt werden, wo der Vater-Prozeß die Tastatur und den Bildschirm behandelt, die eigentliche Editierung aber von **ed** erledigen läßt. Wir haben nicht den Platz, einen Bildschirmeditor vorzuführen, deshalb wird unser Beispiel geradezu lächerlich einfach sein. Es ist ein interaktives Such-Programm mit vielen Ähnlichkeiten zu **grep**. Hier ist das Beispiel einer Sitzung:

[9] Genauer gesagt, als eine *Koroutine*.

```
$ search

File? data

Search pattern? ^a
apple
apricot

Search pattern? apple
apple
pineapple

Search pattern? o$
tomato
mango

Search pattern? EOT
$
```

Die Obstdatei stammt von den Sortierbeispielen weiter oben.

Als wir **sort** benutzten, wußten wir, wann wir mit Lesen der Ausgabe aufzuhören hatten: wenn wir das Dateiende erreichten. Die Situation war einfach, da **sort** seine ganze Eingabe liest, seine ganze Ausgabe schreibt und dann terminiert. Der Editor jedoch ist interaktiv: Er liest etwas Eingabe, kann oder kann auch nicht Ausgabe von unbestimmter Länge schreiben und wartet dann auf weitere Eingaben. Die Ausgabe können wir uns holen, indem wir die Standard-Ausgabe in eine Pipe lenken — doch wie wissen wir, wieviel wir auf einmal lesen sollen? Wir können ja nicht auf *end-of-file* warten, da der Editor seinen Filedeskriptor für die Ausgabe erst bei seiner Termination schließt. Wenn wir zu weit lesen, geraten wir in eine Deadlock-Situation, und lesen wir zu wenig, so verlieren wir die Synchronisation zwischen einem Kommando und dessen Resultaten.

Falls wir den Editor ändern könnten, würden wir ihn eine unzweideutige Datenzeile schicken lassen, wenn er für weitere Eingaben bereit ist. Tatsächlich hat der Editor die Möglichkeit, den Benutzer zur Eingabe aufzufordern (einzuschalten über das **P**-Kommando), doch ist das Aufforderungszeichen * — also kaum unzweideutig. Irgendwie müssen wir den Editor überreden, uns das Ende der Ausgabe des Resultats eines Kommandos mitzuteilen.

Falls es je einen gemeinen Trick gab, dann diesen: Nach jedem Kommando führen wir ein **r**-Kommando (lies Datei) mit einem nichtexistenten Dateinamen aus und warten auf die resultierende Fehlermeldung. Wenn diese Meldung auftaucht, wissen wir, daß der Editor auf unser falsches **r**-Kommando geantwortet hat und

für ein neues Kommando bereit ist. Der nichtexistente Dateiname sollte so gewählt werden, daß die Fehlermeldung wirklich eindeutig ist. Es ist wahrscheinlich am besten, einen Namen zu nehmen, der aus ASCII-Kontrollzeichen besteht, die selten in einem Dateinamen auftauchen, aber aus Gründen der Klarheit nehmen wir den Namen **end-of-file**. Um die genaue Fehlermeldung zu sehen, starten wir den Editor:

```
$ ed
r end-of-file
?end-of-file
q
$
```

So müssen wir nach **?end-of-file** Ausschau halten.

Um die Dinge einfacher zu machen, erstellen wir kleine Unterprogramme, die sich mit dem Editor unterhalten. Am Anfang rufen wir **edinvoke** auf, um den Editor zu starten und die Pipes zu etablieren. Anstatt die Filedeskriptoren direkt zu benutzen, erzeugen wir statt dessen Positionszeiger aus der Standard-Bücherei der Ein- und Ausgabe, indem wir **fdopen** verwenden. Dann können wir zum Editor über **sndfp** schreiben und über **rcvfp** von ihm lesen. Hier ist nun **edinvoke**, das dem ersten Teil von **fsort** gleicht:

```
static FILE * sndfp, * rcvfp;  /* fuer edsnd und edrcv        */

static edinvoke()              /* Start des Editor-Prozesses */
{
    int pfdout[2], pfdin[2];

    if (pipe(pfdout) == -1 || pipe(pfdin) == -1)
        syserr("pipe");
    switch (fork())
    {
        case -1:
            syserr("fork");
        case 0:
            if (close(0) == -1)
                syserr("close");
            if (dup(pfdout[0]) != 0)
                fatal("dup");
            if (close(1) == -1)
                syserr("close");
```

```
            if (dup(pfdin[1]) != 1)
                fatal("dup");
            if (close(pfdout[0]) == -1 || close(pfdout[1]) == -1 ||
                close(pfdin[0]) == -1  || close(pfdin[1]) == -1)
                syserr("close");
            execlp("ed", "ed", "-", NULL);
            syserr("execl");
        }
        if (close(pfdout[0]) == -1 || close(pfdin[1]) == -1)
            syserr("close");
        sndfp = fdopen(pfdout[1], "w");
        rcvfp = fdopen(pfdin[0], "r");
    }
```

Um in die Pipe zu schreiben bzw. aus ihr zu lesen, benützen wir **edsnd**, **edrcv** und **turnaround**. **edrcv** liefert **FALSE**, falls keine weitere Editor-Ausgabe verfügbar ist — dies erkennt **edrcv**, wenn es die erzwungene Fehlermeldung findet. Um sicher zu sein, daß die Fehlermeldung da ist, muß **turnaround** aufgerufen werden, um vom Senden auf Empfang umzuschalten. Hier sind alle drei Funktionen:

```
static void edsnd(s)            /* Zeile zum Editor schicken */
    char * s;
{
    if (fputs(s, sndfp) == EOF)
        fatal("edsnd");
}

static BOOLEAN edrcv(s, max)    /* Zeile vom Editor holen    */
    char * s;
    int max;
{
    if (fgets(s, max, rcvfp) == NULL)
        fatal("edrcv");
    return(strcmp(s, "?end-of-file\n") != 0);
}

static void turnaround()        /* Umschalten von edsnd auf  */
{                               /* edrcv                     */
    edsnd("r end-of-file\n");
    fflush(sndfp);
}
```

Wir leeren **sndfp**, so daß der Editor unsere ganze Ausgabe erhält. Erinnern Sie sich daran, daß die Ausgabe zu Pipes normalerweise durch die Standard-Bücherei der Ein- und Ausgabe gepuffert wird.

Da wir im Normalfall alles sehen wollen, was der Editor uns zu sagen hat, kommt eine neue Funktion ins Spiel:

```
static void rcvall()  /* Editor-Ausgabe empfangen und ausgeben */
{
    char s[200];

    turnaround();
    while (edrcv(s, sizeof(s)))
        printf("%s", s);
}
```

Schließlich kommt noch das Hauptprogramm, das den Dialog mit dem Benutzer erledigt und mit dem Editor in Verbindung tritt:

```
main()  /* interaktives Such-Programm   */
{
    char s[100], line[100];
    BOOLEAN prompt();

    edinvoke();
    if (!prompt("File", s))
        exit(0);
    sprintf(line, "e %s\n", s);
    edsnd(line);
    rcvall();
    while (prompt("Search pattern", s))
    {
        sprintf(line, "g/%s/p\n", s);
        edsnd(line);
        rcvall();
    }
    edsnd("q\n");
    exit(0);
}
```

```
static BOOLEAN prompt(msg, result)  /* Aufforderung an */
    char * msg, * result;           /* den Benutzer    */
{
    printf("\n%s? ", msg);
    return(gets(result) != NULL);
}
```

Ist dieses Programm nun frei von der Gefahr einer Deadlock-Situation? Da wir wissen, daß die Kapazität einer Pipe mindestens 4096 Bytes beträgt, wird der Vater-Prozeß nie blockieren während er in die Eingabe-Pipe des Editors schreibt, da unsere Editor-Kommandos relativ kurz sind. Auch der Editor wird beim Schreiben zum Vater-Prozeß niemals blockieren (höchstens kurzfristig), da der Vater alles liest, was vom Editor kommt. Es gibt auch keinen Grund, warum die Fehlermeldung bezüglich der falschen Datei fehlen könnte, außer jemand erzeugt wirklich eine Datei mit diesem Namen. Dies ist zwar kein Beweis, jedoch werden die offensichtlichen Fallen vermieden.

6.6 Portabilität

Außer den blockierungsfreien **read**- und **write**-Aufrufen ist alles in diesem Kapitel über alle UNIX-Versionen hinweg portabel. Die Portabilität von **fcntl** und der **O_NDELAY**-Flagge wurde in Abschnitt 3.14 diskutiert. Der Gebrauch von **fcntl** anstelle von **dup** ist unnötig, und **O_NDELAY** wird selten für Pipes benötigt. Um also portabel zu sein genügt es, daß die meisten Programme diese Punkte einfach überspringen.

Der **pipe**- und der **dup**-Systemaufruf funktionieren in allen UNIX-Versionen so, wie wir sie beschrieben haben. Tatsächlich kann kein Betriebssystem von sich behaupten, es sei UNIX, wenn diese Aufrufe nicht wie gewohnt funktionieren.

Version 7 und 4.2 BSD unterstützen eine Version von **dup**, nämlich **dup2**, die in folgendem Punkt ähnlich zu **fcntl** ist: Man kann die Nummer des Filedeskriptors für das Duplikat angeben. Ein Unterschied ist allerdings, daß dieser Filedeskriptor, falls nötig, geschlossen wird, um zu garantieren, daß er anschließend auch benutzt wird. Jedoch ist **dup2** ebenso entbehrlich wie **fcntl**, und da dieser Systemaufruf außerdem in System III und in System V nicht vorhanden ist, sollte man auf seine Verwendung verzichten.[10]

[10] Die Entwickler von System III hatten Recht, daß sie **dup2** als *Systemaufruf* entfernten, als der allgemeinere Systemaufruf **fcntl** dazugefügt wurde. Sie hätten jedoch aus Gründen der Aufwärtskompatibilität zu Version 7 ein **dup2**-*Unterprogramm* zur Verfügung stellen sollen.

Aufgaben

6.1 Benutzen Sie die Standard-Shell (**sh**) oder die Berkeley-Shell (**csh**), tippen Sie ungültige Kommandozeilen ein und sehen Sie, welche Fehlermeldungen Sie erhalten, falls Sie überhaupt welche erhalten. Probieren Sie für den Anfang folgendes aus:

```
echo abc >f1 >f2
echo def | cat <f1
echo ghi >f1 | cat
cat <f1 <f2
echo jkl | | cat
echo jkl | >f1
```

6.2 Programmieren Sie **dup** mit Hilfe von **fcntl**.

6.3 In der Funktion **gettoken** in Abschnitt 6.4 wird im Zustand **INQUOTE**, wenn **getchar** aufgerufen wird, nicht auf *end-of-file* geprüft. Kann dies zu Schwierigkeiten führen? Weshalb bzw. weshalb nicht?

6.4 Bringen Sie der Shell aus Abschnitt 6.4 Parameter-Ersetzung (wie z.B. in `echo $PATH`) bei.

6.5 Bringen Sie der Shell aus Abschnitt 6.4 die Erzeugung von Dateinamen (sogenannte *wild cards*) bei. Beschränken Sie sich auf den letzten Teil eines Pfadnamens (d.h. `ls */*.c` soll unzulässig sein).

6.6 Wie Aufgabe 6.5, aber ohne die genannte Einschränkung. Hinweis: Verwenden Sie Rekursion.

6.7 Die Shell aus Abschnitt 6.4 macht alle Prozesse in einer Pipeline zu ihren Söhnen. Die Standard-Shell (**sh**) macht jeden Prozeß zum Sohn des Prozesses auf seiner Rechten, und nur der am weitesten rechts stehende Prozeß ist ein Sohn der Shell. Modifizieren Sie die Shell aus Abschnitt 6.4 dahingehend, daß die Vaterschaftsverhältnisse die gleichen wie in der Standard-Shell sind.[11]

6.8 Fügen Sie der Shell aus Abschnitt 6.4 eine **goto**-Anweisung hinzu.

6.9 Fügen Sie der Shell aus Abschnitt 6.4 eine **if**-Anweisung hinzu.

[11] In neueren UNIX-Versionen, wie z.B. System V.2 oder Berkeley 4.3, verhält sich **sh** anders als hier beschrieben. Dort ist das am weitesten rechts stehende einfache Kommando einer Pipeline ein direkter Abkömmling der gerade laufenden Shell. Alle anderen Kommandos der Pipeline werden als Sohn-Prozesse dieses Prozesses erzeugt, d.h., sie sind Enkel der Shell. A.d.Ü.

6.10 Entwerfen und entwickeln Sie eine sogenannte *Menü-Shell.* Diese Shell fordert nicht zur Kommandoeingabe auf, sondern zeigt statt dessen eine Auswahl an Möglichkeiten, von denen der Benutzer einfach eine auswählt. Sie benötigen außerdem ein Menü für Argumente. Anstatt die Einzelheiten der verschiedenen Kommandos in die Shell einzubauen, sollte die Shell ihre Informationen über Kommandos und deren Argumente aus einer Datenbank holen.

6.11 Planen und entwickeln Sie ein sogenanntes *Window-System.* Sie können dafür das **curses**-Paket benützen, falls es Ihnen zur Verfügung steht. Teilen Sie den Schirm in zwei Hälften (was ist einfacher — horizontale oder vertikale Aufteilung?). Lassen Sie in jedem dieser Fenster eine Shell laufen. Durch Drücken einer Funktionstaste wählt der Benutzer das aktive Fenster aus. Ein Kommando das im inaktiven Fenster läuft, fährt mit der Ausgabe auf den Schirm fort, doch wenn Eingabe verlangt wird, blockiert es solange, bis der Benutzer dieses Fenster aktiviert und eine Antwort eingibt. Ist ein Fenster voll, so wird es weitergerollt; der Benutzer kann es jedoch (bis zu einer gewissen Grenze) zurückrollen um nachzusehen, was verschwunden ist. Hinweis: Der schwierigste Teil daran ist, zu verhindern, daß die Ausgabe eines Fensters das andere Fenster überschreibt. Entwurfsfrage: Sollte die Standard-Ausgabe eines Kommandos direkt an das Terminal-Gerät gehen, durch einen speziell dafür entworfenen Filter fließen oder an die Shell des Fensters zurückgehen?

6.12 Nachdem Sie Aufgabe 6.11 beendet und zur Perfektion gebracht haben, verkaufen Sie die Lösung an einen Software-Händler. Benützen Sie Ihre Tantiemen, um weitere Exemplare dieses Buchs zu kaufen.

6.13 Schreiben Sie ein Programm, das seine Prozeßnummer auf die Standard-Ausgabe schreibt und dann eine Liste von Prozeßnummern von der Standard-Eingabe liest. Ist die eigene Prozeßnummer auf der Liste, so soll es die Liste (mit Hilfe des Filedeskriptors 2) ausgeben. Andernfalls fügt es seine Prozeßnummer zur Liste hinzu und schreibt die gesamte Liste auf die Standard-Ausgabe. Danach fängt es wieder von vorne an. Installieren Sie fünf Prozesse, die dieses Programm in einem Ring laufen lassen und lassen Sie sie eine Weile herumspielen (dies ist ein Computerspiel, das ein Computer mit sich selbst spielen kann).

6.14 Schreiben Sie für den Taschenrechner **dc** eine neue Benutzerschnittstelle, die es dem Benutzer erlaubt, Ausdrücke in Präfix-Notation (z.B. `2 + 3`) anstelle von Postfix-Notation (z.B. `2 3 +`) einzugeben (so wie es das **bc**-Kommando macht.) Verwenden Sie hierfür eine ähnliche Technik wie das interaktive Suchprogramm aus Abschnitt 6.5.

Kapitel 7: Fortgeschrittene Prozeßkommunikation

7.1 Einführung

1973, nur ein paar Jahre nachdem UNIX erfunden wurde, begannen mehrere Abteilungen der Bell Laboratorien UNIX als Grundlage für Systeme zur Transaktionsverarbeitung zu benutzen. Diese Systeme, die zur Automation der Verwaltung von Telefongesellschaften und der Buchhaltung gebaut wurden, bestanden aus Terminals, die auf ein Datenbank-Managementsystem zugriffen, das auf einem PDP-11 Minicomputer lief. Seit damals wurden Hunderte dieser sogenannten arbeitsunterstützenden Systeme installiert — diese waren früher unter dem Namen *Bell-System* bekannt.

Schon von Beginn an erkannten die Architekten dieser Systeme, daß UNIX wesentliche Fähigkeiten in bezug auf Datenbank-Management fehlten, vorwiegend im Bereich der Prozeßkommunikation. Da damals UNIX sogar noch kleiner und einfacher war als heute, fügten Systemprogrammierer, die in Entwicklungsgruppen für unterschiedliche Anwendungen tätig waren, einfach das, was sie brauchten, zum Kern hinzu. So entstand bald eine Version von UNIX, die für Datenbank-Management gut geeignet war. Sie enthielt Systemaufrufe zur Unterstützung von Messages, Semaphoren und Shared-Memory. Dieses System hieß Columbus UNIX, benannt nach der Stadt in Ohio, wo die Entwickler arbeiteten (in einer Filiale des Unternehmens). In den späten 70er Jahren war dieses System bereits ein interner Standard für arbeitsunterstützende Systeme, obwohl es bis dahin außerhalb der Bell Laboratorien beinahe unbekannt war. Die extern freigegebenen Versionen (bis einschließlich System III) wurden intern nicht für ernsthaftes Datenbank-Management benutzt, und es herrschte wenig Interesse, sie für solche Aufgaben anzupassen, da Columbus UNIX die bevorzugte Alternative war.

Um die Prozeßkommunikation für Datenbank-Managementsysteme geeigneter zu machen, erweiterten die Entwickler System III zwar um FIFO-Dateien, die Entwickler der arbeitsunterstützenden Systeme zeigten sich davon aber unbeeindruckt.

Diese blieben an Columbus UNIX kleben, das vertrauter und viel schneller war. Erst mit System V wurden in den extern freigegebenen Versionen Möglichkeiten zur Prozeßkommunikation eingebaut, die denen von Columbus UNIX ähnlich waren. Nun war das System zum ersten Mal zu schneller Transaktionsverarbeitung fähig. Ironischerweise sieht es für die außenstehende Welt so aus, als ob Messages, Semaphore und Shared-Memory ungefähr drei Jahre nach den FIFO-Dateien erschienen, tatsächlich gab es sie aber bereits fünf Jahre davor.

Wir beginnen unsere Untersuchung dieser fortgeschrittenen Möglichkeiten der Prozeßkommunikation, indem wir einige Punkte des Systementwurfs von Datenbank-Managementsystemen einführen, die Prozeßkommunikation betreffen — diese wurden ursprünglich genau für diesen Zweck hinzugefügt. Wir werden als nächstes zeigen, wie man FIFO-Dateien benützt, die in allen UNIX-Versionen vorhanden sind, die von System III abgeleitet sind (einschließlich System V, Xenix 3 und PC/IX). Danach werden wir Messages, Semaphore, Shared-Memory und Record-Locking betrachten: Messages, wie in System V implementiert; Semaphore und Shared-Memory, wie in System V und Xenix 3 implementiert; Record-Locking, wie in Xenix 3 und PC/IX implementiert und wie auch von der Benutzergruppe */usr/group* vorgeschlagen.

Eines unserer Ziele in diesem Kapitel ist es, einen Satz einfacher Werkzeuge zur Prozeßkommunikation zu entwickeln, der auf mehreren UNIX-Versionen implementiert werden kann, und der jeweils die besonderen Leistungsmerkmale jeder Version ausnützt. Zum Beispiel werden wir eine Menge an Funktionen einführen, die Messages senden und dafür FIFO-Dateien benutzen. Anschließend werden wir diese Funktionen neu bauen, indem wir die Systemaufrufe von System V benutzen, die sich mit Messages befassen. Dieses Vorgehen wird nicht alle Möglichkeiten der Systemaufrufe von System V ausnutzen (die wirklich phantastisch sind), aber es erlaubt die freie Portierung von Datenbank-Managementsystemen zwischen diesen Versionen. Gleichzeitig wird so die erhöhte Leistungsfähigkeit von System V ausgenützt.

7.2 Feinheiten eines Datenbank-Managementsystems

Für unsere Zwecke definieren wir ein *Datenbank-Managementsystem* (DBMS) als ein Softwaremodul, das das Informationsmodell eines Unternehmens unterhält und den Zugriff darauf regelt. Ein DBMS besteht im wesentlichen aus zwei Teilen: der bleibenden *Datenbasis*, die in Dateien gehalten wird (unterteilt in Datensätze und Felder, mit einem Index für schnellen Zugriff versehen), und Routinen zum Zugriff, die es anderen Modulen (Anwendungsprogrammen) erlauben, Daten zu speichern, wiederzufinden, zu ändern und zu löschen. Ein Anwendungsprogramm, das dieses DBMS benutzt, wird *Klient* (*client*) genannt.

In einem System zur Transaktionsverarbeitung wird eine Maske auf dem Bildschirm eines Terminals angezeigt, ein Sachbearbeiter füllt die freien Stellen aus und die vervollständigte Maske wird von einem Anwendungsprogramm bearbeitet, das auf das DBMS zugreift, um die geforderte Aufgabe durchzuführen. Die Ergebnisse werden wieder auf dem Schirm dargestellt.

Möglicherweise greifen viele Benutzer über viele Terminals gleichzeitig auf die gleiche Datenbank zu. Es kann zwar für jeden Benutzer einen eigenen Anwendungsprozeß und eine eigene Kopie der Zugriffsroutinen geben — da es die Datenbasis selbst nur einmal geben kann, muß diese gemeinsam benutzt werden. Deshalb ist das Kontrollieren des gleichzeitigen Zugriffs auf die Daten eine wichtige Funktion des DBMS, damit die Aktionen eines Benutzers nicht denen eines anderen in die Quere kommen.

Wir verschieben eine ausführlichere Diskussion über die Kontrolle von gleichzeitigen Zugriffen bis Abschnitt 7.8. Im Moment nehmen wir einfach an, daß ein Teil des DBMS ein sogenannter *Lock-Manager* ist, der für jeden Datensatz, der von einem Prozeß benutzt wird, eine Sperre verwaltet. Ein zweiter Prozeß kann auf einen gesperrten Datensatz solange nicht zugreifen, bis ihn der erste Prozeß freigibt. So wie es nur eine Ausführung der Daten gibt, so kann es auch nur einen Lock-Manager geben. Zu der Frage, wo sich dieser befindet, kommen wir in Bälde. Beachten Sie jedoch, daß der Lock-Manager nicht einfach eine weitere gemeinsame Datei sein kann, die die eigentlichen Sperren enthält, da dann auf diese Datei gleichzeitig zugegriffen würde und man einen Weg finden müßte, *diese* zu sperren. In anderen Worten: Der Zugriff auf den Lock-Manager muß über einen anderen Mechanismus als der des Lock-Managers selbst kontrolliert werden.

Es gibt zwei Arten, die Zugriffsroutinen zu gliedern. Bei der *dezentralisierten* Methode stellen sie einfach eine Bücherei von Unterprogrammen dar, die in jedes Anwendungsprogramm mit eingebunden wird. Jeder Anwendungsprozeß hat somit eine eigene Kopie. Bei der *zentralisierten* Methode existieren die Zugriffsroutinen in einem einzigen DBMS-Prozeß; ein Anwendungsprozeß führt eine Routine aus, indem er dem zentralen DBMS-Prozeß eine Nachricht schickt. Daraufhin wird eine Nachricht zurückgeschickt, die die Antwort enthält (ein Status-Code oder einige Datensätze). Um mit der zentralisierten Methode einen guten Durchsatz zu erreichen, muß die Prozeßkommunikation schnell sein.

Falls sie durchführbar ist, ist die zentralisierte Methode besser. Es gibt nur eine Ausführung der Zugriffsroutinen, was sowohl Speicher als auch Zeit beim Ein- und Auslagern von Prozessen spart. Dadurch daß das DBMS ein separater Prozeß ist, wird dieses vor Beschädigungen geschützt, die durch ein fehlerhaftes oder Abwege gehendes Anwendungsprogramm hervorgerufen werden könnten. Es kann auch mit einer eigenen Benutzernummer ausgeführt werden, so daß das Zugriffsschutz-System in UNIX die Daten schützt. Dadurch, daß nur ein Prozeß direkten Zugriff auf die Daten hat, kann dieser schließlich leicht gleichzeitigen Zugriff kontrollieren, indem er eine Tabelle der Sperren in seinem eigenen Adreßraum hält — kein Sperrmecha-

nismus des Kerns ist nötig oder wünschenswert. Es gibt auch keinen gleichzeitigen Zugriff auf diese Tabelle, da nur ein Prozeß mit ihr arbeitet.

Die dezentralisierte Methode ist etwas einfacher zu verstehen — eine Bücherei voller Unterprogramme ist etwas Alltägliches. Da keinerlei Prozeßkommunikation nötig ist folgt, daß diese Methode potentiell schneller und portabler ist. Allerdings kann der Lock-Manager nicht dezentralisiert werden, woraus sich ein ernstes Hindernis in der Anwendung dieser Methode ergibt. Mehrere Lösungen sind denkbar:

- Verwenden Sie die "Kopf-in-den-Sand"-Technik: Fahren Sie das System ohne Sperren. Wenn sich Benutzer über seltsame Resultate beschweren, tun Sie so, als ob Sie sie nicht hören oder schieben Sie es auf die Hardware.[1]
- Implementieren Sie den Lock-Manager als zentralisierten Prozeß und verwenden Sie Messages zur Prozeßkommunikation. Natürlich können Sie, wenn Sie sich diese Mühe machen, auch gleich das ganze DBMS zentralisieren.
- Implementieren Sie einen unechten Gerätetreiber, der als Lock-Manager fungiert. Das ist leichter, als den Kern zu ändern, und Sie brauchen dazu auch nicht den Quelltext des Kerns. Benützen Sie den Systemaufruf **fcntl**, um Sperren zu errichten und zu löschen.
- Benutzen Sie eine UNIX-Version, die bereits einen Lock-Manager im Kern eingebaut hat, wie etwa Xenix 3 oder PC/IX.

Keine dieser Alternativen ist völlig zufriedenstellend. Das Implementieren eines Gerätetreibers hört sich nach viel Arbeit an, was es auch ist. Besitzt Ihre Version des UNIX-Kerns bereits einen Lock-Manager, so ist es wahrscheinlich leicht, diesen zu benutzen — allerdings wird Ihr System so nicht portabel sein. Weiterhin kann der eingebaute Lock-Manager zu langsam sein; für Systeme, die einen hohen Durchsatz erfordern, muß der Lock-Manager oft zusammen mit dem Rest der Software für die Datenbank entwickelt werden.

Zusammenfassend empfehle ich die zentralisierte Methode, da diese Speicher spart und verläßlicher und sicherer ist. Sie kann mittels FIFOs *oder* Messages implementiert werden, und sie benötigt keine durch den Kern bereitgestellten Sperrmechanismen.[2]

[1] Lachen Sie nicht — dies ist verbreiteter, als wir zugeben würden!

[2] Eine verbreitete Kritik an UNIX ist das Fehlen von Record-Locking. Zum Beispiel schrieb ein Journalist in einer Einzelausgabe von *Computerworld Extra* (26. September 1984), daß "ein wesentlicher Beitrag der Anstrengungen von */usr/group* auf Einstimmigkeit bezüglich eines Standards für (Record-)Locking beruht. ... Diese Fähigkeit wird bei kommerziellen Mehrbenutzer-Anwendungen entscheidend benötigt. ..." Ein anderer Journalist schrieb, daß "Fähigkeiten wie Record- und File-Locking ... für kommerzielle Betriebssysteme allgemein als notwendig angesehen werden." Diese letzte Aussage ist bemerkenswert, wenn man bedenkt, daß das Betriebssystem von IBM (MVS), sicherlich das erfolgreichste kommerzielle Betriebssystem, Record-Locking *nicht* unterstützt — dies wird durch das zentralisierte DBMS, in der Regel IMS, zur Verfügung gestellt. Über ein Dutzend

Da wir jetzt einige Grundlagen des DBMS verstehen, können wir fortgeschrittene Möglichkeiten der Prozeßkommunikation darstellen und ihre Verwendung in der Implementierung eines DBMS betrachten.

7.3 FIFO-Dateien oder Pipes mit Namen

Eine FIFO-Datei vereinigt die Fähigkeiten einer Datei und einer Pipe. Wie eine Datei hat sie einen Namen, und jeder Prozeß mit passenden Rechten darf sie zum Lesen oder Schreiben öffnen. Anders als bei einer Pipe können aber auch Prozesse, die nicht miteinander verwandt sind, über eine FIFO-Datei miteinander kommunizieren — die Prozesse müssen sich nicht allein auf eine Erbschaft verlassen, um auf eine FIFO-Datei zuzugreifen. Ist eine FIFO-Datei jedoch einmal geöffnet, so verhält sie sich mehr wie eine Pipe als wie eine Datei. Geschriebene Daten werden in *First-In-First-Out*-Manier gelesen, und einzelne Systemaufrufe zum Lesen oder Schreiben sind garantiert atomar, vorausgesetzt, daß die gelesene oder geschriebene Menge nicht die Kapazität der FIFO-Datei übersteigt. Die Kapazität ist die gleiche wie bei einer Pipe (implementierungssabhängig, jedoch mindestens 4096 Bytes). Daten, die einmal gelesen sind, können weder nochmals gelesen werden, noch funktioniert **lseek**.

Wenn eine FIFO-Datei zum Lesen geöffnet wird, so wartet **open** normalerweise, bis sie auch zum Schreiben geöffnet wird, was in der Regel durch einen anderen Prozeß geschieht. Ähnlich blockiert auch ein Öffnen zum Schreiben die FIFO-Datei, bis diese zum Lesen geöffnet wird. Welcher Prozeß **open** daher zuerst ausführt, egal ob lesend oder schreibend, wird auf den anderen warten. Dies erlaubt es den Prozessen, sich selbst zu synchronisieren, bevor die eigentliche Datenübertragung beginnt.

Ist bei **open** die **O_NDELAY**-Flagge gesetzt, so kehrt ein Öffnen zum Lesen sofort zurück — auf den Sender wird nicht gewartet. Umgekehrt erzeugt Öffnen zum Schreiben einen Fehler, falls kein Empfänger die FIFO-Datei geöffnet hat. Diese Asymmetrie impliziert, daß **O_NDELAY** beim Öffnen zum Lesen nützlich, beim Öffnen zum Schreiben jedoch ungeschickt ist. Dies liegt daran, daß der Fehler dann bearbeitet und eventuell **open** noch einmal versucht werden muß. Der Sinn dieser Asymmetrie liegt darin, einen Prozeß daran zu hindern, Daten in eine FIFO-Datei zu schreiben, die nicht unmittelbar danach gelesen werden, da es in UNIX keine Möglichkeit gibt, Daten permanent in FIFO-Dateien zu speichern. Es ist wie bei einer Wasserleitung: Man hat eigentlich kein Recht, das Wasser anzudrehen, bevor nicht beide Enden verlötet sind. Sind noch Daten in einer FIFO-Datei, wenn

Jahre Erfahrung der Bell Laboratorien in der Implementierung von Systemen zur Transaktionsverarbeitung mit hohem Durchsatz hat gezeigt, daß Record-Locking auch in UNIX unnötig ist. Für den Kern ist es etwa genauso wichtig, Record-Locking zu unterstützen wie etwa trigonometrische Funktionen.

alle Empfänger und Sender ihre Filedeskriptoren schließen, so werden die Daten ohne Fehleranzeige ganz einfach weggeworfen. Auch das ist wieder wie bei der Wasserleitung: Wasser läuft aus, wenn man das Rohr an beiden Verbindungen löst.

Die **O_NDELAY**-Flagge beeinflußt **read** und **write** bei FIFO-Dateien genauso wie bei Pipes. Das bedeutet, falls die Flagge nicht gesetzt ist, blockiert der Empfänger, bis Daten verfügbar sind, und der Sender blockiert, wenn die FIFO-Datei voll wird. Ist die Flagge dagegen gesetzt, so blockiert keiner der beiden; statt dessen wird eine Fehlermeldung geliefert.

In Abschnitt 3.8 zeigten wir, wie man **mknod** aufruft um eine FIFO-Datei zu erzeugen, und in Abschnitt 3.12 zeigten wir, wie man sie mit **stat** und **fstat** befragt.

Eine offensichtliche Anwendung einer FIFO-Datei besteht in der Verwendung als Ersatz für eine Pipe. Anstatt den Systemaufruf **pipe** zu benutzen, erzeugt man stattdessen eine FIFO-Datei und öffnet sie dann zweimal, um Filedeskriptoren zum Lesen und Schreiben zu erhalten. Von da an kann man eine FIFO-Datei wie eine Pipe behandeln. Benutzt man FIFO-Dateien jedoch wirklich nur auf diese Weise, so haben sie überhaupt keine Vorteile gegenüber Pipes, doch dafür mehrere Nachteile: Extrakosten beim Erzeugen der FIFO-Datei, zwei Systemaufrufe, um die Filedeskriptoren zu erhalten und das Risiko eines Namenkonflikts (wie bei temporären Dateien).

FIFO-Dateien wurden zu UNIX nicht als Ersatz für *Pipes* hinzugefügt — sie wurden hinzugefügt, um *Messages* zu ersetzen, da viele Systemprogrammierer innerhalb der Bell Laboratorien darauf bestanden, daß diese für eine effiziente DBMS-Implementierung nötig seien. Tatsächlich war die Idee der benannten Pipes seit den frühen 70er Jahren wohlbekannt, blieb aber bis 1978 unbeachtet, als System III entwickelt wurde. Die Entwickler von System III merkten, daß Messages zum Einbau in UNIX ungeeignet waren, wohingegen FIFO-Dateien leicht, effektiv und elegant waren. Jeder, der die Handbuchseiten von System V über Messages liest, wird geneigt sein, dem zuzustimmen.

7.4 Implementierung von Messages mittels FIFO-Dateien

Einfache FIFO-Dateien funktionieren nicht gut als Träger für Messages — ein zusätzlicher Mechanismus muß hinzugefügt werden. Wir werden dies zeigen, indem wir zwei Routinen implementieren, nämlich **send** und **receive**. Dann werden wir diese Routinen in mehreren Anwendungsbeispielen benützen, die in einem zentralisierten DBMS gipfeln, das mehrere Klienten behandeln kann.

Hier sind die Deklarationen von **send** und **receive**:

```
BOOLEAN send(dstkey, buf, nbytes)  /* Message schicken              */
    long dstkey;                   /* Schluessel des Ziels         */
    char * buf;                    /* Zeiger auf Message-Daten     */
    int nbytes;                    /* Zahl der Bytes in der Message */

/* liefert TRUE bei Erfolg oder FALSE bei Fehler                   */

BOOLEAN receive(srckey, buf, nbytes)  /* Message empfangen         */
    long srckey;                      /* Schluessel der Quelle     */
    char * buf;
    int nbytes;

/* liefert TRUE bei Erfolg oder FALSE bei Fehler                   */
```

Messages werden an eine *Queue* (*Warteschlange*) geschickt, die durch den *Schlüssel*, eine long-Integer, identifiziert wird. **buf** zeigt auf die Message, die **nbytes** Bytes lang ist. Beachten Sie, daß die Argumente denen von **read** und **write** ähneln, was beabsichtigt ist.

Messages, die an eine Queue geschickt werden, werden dort in der Reihenfolge ihrer Ankunft abgelegt. Eine Message darf beliebige Daten beinhalten, muß jedoch mit einer long-Integer beginnen, die im Moment nicht benutzt wird. Eine Message könnte also wie folgt strukturiert sein:

```
struct msgdata
{
    long unused;
    char data[100];
};
```

Dies gestattet einer Message, 100 Bytes in Anspruch zu nehmen. Im Augenblick brauchen wir die long-Integer am Anfang nicht — wir werden sie in Abschnitt 7.5 verwenden, wenn wir **send** und **receive** mit den Messages aus System V nachbauen. Im Programmkopf ist **buf** als Zeiger auf **char** anstatt als Zeiger auf **struct msgdata** deklariert. Wie wir noch sehen werden, liegt dies daran, daß sich die tatsächliche Struktur je nach Anwendung ändert.

Die Queue kann auch voll werden. Wenn dem so ist, blockiert **send** solange, bis die Queue soweit geleert ist, daß genügend Platz für die zu sendende Message vorhanden ist.

receive nimmt die älteste Message aus der Queue heraus und kopiert diese in die Speicherfläche, auf die **buf** verweist. Dabei werden höchstens **nbytes** kopiert. Übriggebliebene Bytes können entweder in der Queue belassen oder weggeworfen werden, je nach Ermessen des Entwicklers (wiederum legen wir uns noch nicht fest, so können wir später immer noch anders implementieren). So oder so ist es

jedoch ein schwerer Fehler, wenn das Argument **nbytes** für **receive** nicht exakt mit dem korrespondierenden Argument des Aufrufs von **send** übereinstimmt, der die Message ablegte. Entweder muß man alle Messages gleich groß machen (was sehr empfehlenswert ist), oder man muß sich etwas einfallen lassen, daß der Empfänger die Größe der Message am Anfang der Queue herausbekommen kann.

Damit zwei Prozesse über **send** und **receive** miteinander kommunizieren können, müssen sie den Empfangsschlüssel der Queue des jeweils anderen kennen. Dann kann jeder Messages an die andere Queue schicken. Die beiden Prozesse könnten sich auch mit nur einer Queue verständigen, doch müssen sie sich dann selbst synchronisieren, um die Probleme aus Abschnitt 6.5 zu vermeiden. (Dort vermieden wir diese Probleme, indem wir anstatt einer zwei Pipes für die Zwei-Weg-Kommunikation verwendeten). Im Falle eines zentralisierten DBMS und seiner Klienten sollte der Schlüssel der Queue des DBMS im voraus eingeführt und jedem Klienten mitgeteilt werden — eine Environment-Variable (etwa **DBMSKEY**) ist ideal für diesen Zweck. Jeder Klient erzeugt seinen eigenen Schlüssel, der einfach aus der Prozeßnummer bestehen kann, da dies Eindeutigkeit garantiert. Mit jeder Anforderung an das DBMS teilt ein Klient seinen Schlüssel mit, so daß das DBMS weiß, an welche Queue es antworten muß. Dies ist ganz analog zu einer Bestellung aus einem Versandhauskatalog: Mit der Anschrift der Firma wird so auffallend geworben, daß jeder einen Brief an diese Firma schicken kann; eine Antwort ist allerdings nur möglich, wenn man seine eigene Adresse beifügt.

Nun zur Implementierung von **send** und **receive** mittels FIFO-Dateien: Eine Queue ist eine FIFO-Datei, deren Namen wir konstruieren, indem wir den Schlüssel in eine Zeichenkette verwandeln und **/tmp/fifo** voranstellen. Zum Beispiel wäre die zum Schlüssel 12345 korrespondierende FIFO-Datei **/tmp/fifo12345**.

Unsere Implementierung muß zwei größere Probleme vermeiden. Erstens wollen wir einen Sender nicht permanent blockieren, falls der erwartete Empfänger seine FIFO-Datei nie zum Lesen öffnet. Falls ein Klient abnormal terminiert, könnte dies das DBMS verklemmen, so daß dieses damit den anderen Klienten nicht mehr zur Verfügung stände. Allerdings wollen wir auch nicht, daß der Sender zu früh aufgibt, falls der Empfänger noch nicht bereit ist, da wir beim Starten der Anwendung mehrere Prozesse (das DBMS und seine anfänglichen Klienten) erzeugen müssen, und es vielleicht ein paar Sekunden dauert, bis alle Teilnehmer in Fahrt kommen. Deshalb setzt ein Sender beim Öffnen einer FIFO-Datei zum Schreiben die **O_NDELAY**-Flagge. Geht dies schief, weil kein Empfänger die FIFO-Datei zum Lesen geöffnet hat, so legt sich der Sender eine Weile schlafen und versucht es dann noch einmal; nach einer gewissen Anzahl Versuche gibt er allerdings auf. Die symbolische Konstante **NAPTIME** entspricht der Anzahl zu schlafender Sekunden und **MAXTRIES** ist gleich der Anzahl der Versuche (so haben wir uns auch in Abschnitt 2.5 verhalten).

Wir kümmern uns deutlich weniger darum, ob der Empfänger blockiert. Vom DBMS erwarten wir, daß es geduldig wartet, wenn es keine Klienten hat — entweder

taucht bald ein Klient auf, oder es gibt eben keine Arbeit, dann spielt Blockieren auch keine Rolle. Ein Klient darf ruhig blockieren, während er auf die Antwort des DBMS wartet — wenn dieses nicht richtig funktioniert, muß der Fehler durch die Verwalter des DBMS behoben werden, und die Klienten sind ohne DBMS sowieso nutzlos. Natürlich treffen diese Annahmen nicht in jeder Situation zu. Wenn Sie wollen, können Sie **receive** leicht so ändern, daß das gleiche Blockierungsverfahren wie bei **send** benützt wird.

Das zweite größere Problem ist, daß einem DBMS mit vielen Klienten leicht die Filedeskriptoren ausgehen können (20 je Prozeß). Das DBMS besitzt drei Standard-Filedeskriptoren (0, 1 und 2), einen für die eigene Queue zum Empfang, einen für jede Queue eines Klienten und einen für jede Datei mit Daten als Inhalt. Nun hat ein DBMS oft mehr als ein Dutzend Dateien und mehrere Dutzend Klienten zu verwalten. Da die FIFO-Dateien aller Klienten nicht gleichzeitig geöffnet sein können, ist die offensichtliche Lösung, eine FIFO-Datei aufzumachen, die Message zu schreiben und dann die FIFO-Datei zu schließen, um den Filedeskriptor wieder verfügbar zu machen. **open** dauert jedoch zu lang, so daß dies die Kommunikation zu sehr verlangsamen würde. Schließlich spielt die Leistungsfähigkeit bei einem DBMS gerade dann eine große Rolle, wenn es viele Klienten besitzt.

Eine gute Lösung ist, ein ähnliches Schema wie bei einem System mit virtueller Speicherverwaltung zu benutzen. Man hält eine begrenzte Zahl an FIFO-Dateien geöffnet, zum Beispiel sieben. Wird eine achte FIFO-Datei benötigt, so schließt man eine geöffnete und benützt ihren Filedeskriptor. Die hierfür (zum Schließen) geeignetste ist die, die man erst am spätesten wieder benötigt (dies kann nachgewiesen werden). Unglücklicherweise kann ein DBMS aber nicht in die Zukunft sehen (wenn dem so wäre, so könnte das DBMS auch gleich jedem Klienten all seine Daten übermitteln). Wenn man von der Theorie ausgeht, daß eine FIFO-Datei, die schon eine Zeitlang nicht mehr benutzt wurde, wahrscheinlich auch eine weitere Weile nicht benutzt wird, so besteht ein vernünftiger Kompromiß darin, die FIFO-Datei zu schließen, die am längsten nicht mehr benutzt wurde. Machen die Klienten ihre Anfragen reihum (nach der sogenannten *round-robin*-Methode), so wird unser Schema versagen, da dann die am längsten nicht mehr benutzte FIFO-Datei gerade die falsche zum Schließen ist. Tendiert allerdings jeder Klient dazu, seine Anfragen zu bündeln, so wird dieses Schema gut funktionieren. Während in einem System zur Transaktionsverarbeitung der Bediener die Maske ausfüllt, bleibt der Klient untätig — danach richtet er in rascher Folge mehrere Anfragen an das DBMS, wiederum gefolgt von einer Zeit des Nichtstuns, in der der Bediener über die Resultate grübelt und mit der nächsten Maske beginnt. Deshalb können wir hoffen, daß diese Am-längsten-nicht-mehr-benutzt-Methode gut funktionieren wird.

Wir können eine Tabelle der aktiven Filedeskriptoren wie folgt verwalten:

```
static struct
{
    long key;
    int fd;
    int time;
} fifos[MAXOPEN];
```

Wann immer wir eine FIFO-Datei lesen oder schreiben, notieren wir die aktuelle Zeit in der Komponente **time** des entsprechenden Elements aus **fifo**. Wenn wir uns eines Filedeskriptors bemächtigen müssen, so durchsuchen wir einfach den Vektor nach dem Element mit der ältesten Zeitangabe. Dann schließen wir die Komponente **fd**, öffnen die neue FIFO-Datei und speichern Schlüssel, Filedeskriptor und aktuelle Zeit in diesem Element.

Da wir die **time**-Komponente nur zum Merken von relativen Zeitangaben benutzen, müssen wir eigentlich nicht die Uhr des Computers befragen, um die aktuelle Zeit zu speichern. Wir können jedesmal, wenn **send** oder **receive** aufgerufen wird, eine lokale Variable (**clock**) hochzählen und diesen Wert dann in der **time**-Komponente speichern. Hat die Uhr nur 16 Bits, so wird sie nach 32767[3] Einheiten überlaufen. Danach gibt es eine kurze Zeitspanne mit nicht ganz optimaler Auswahl, der jedoch schnell wieder eine vernünftige Auswahl folgt. Dies ist vollkommen akzeptabel, da die Am-längsten-nicht-mehr-benutzt-Methode von vorneherein unausgewogen ist. Eine Uhr mit mehr Bits wäre zuviel des Guten.

Hier ist der Text von **send** und **receive**. Wir haben den Aufruf **rmqueue** hinzugefügt, um eine Queue freizugeben, wenn diese nicht länger benötigt wird (bevor ein Klient terminiert). Die Funktion **openfifo** verwendet das Am-längsten-nicht-mehr-benutzt-Schema, um einen unbenutzten Filedeskriptor zu finden, und öffnet diesen dann mit dem Warteverfahren, das wir weiter oben diskutierten.

```
#define MAXOPEN 7
#define MAXTRIES 3
#define NAPTIME 5

static char * fifoname(key)          /* Name der FIFO-Datei aus    */
    long key;                        /* Schluessel erzeugen        */
{
    static char fifo[20];

    sprintf(fifo, "/tmp/fifo%ld", key);
    return(fifo);
}
```

[3] $32767 = 2^{15} - 1$. A.d.Ü.

```
static int openfifo(key, flags)        /* Filedeskriptor der          */
    long key;                          /* FIFO-Datei liefern          */
    int flags;
{
    static struct
    {
        long key;
        int fd;
        int time;
    } fifos[MAXOPEN];
    static int clock;
    int i, avail, oldest, fd, tries;
    char * fifo;
    extern int errno;

    clock++;
    avail = -1;
    for (i = 0; i < MAXOPEN; i++)
    {
        if (fifos[i].key == key)
        {
            fifos[i].time = clock;
            return(fifos[i].fd);
        }
        if (fifos[i].key == 0 && avail == -1)
            avail = i;
    }
    if (avail == -1)                   /* alle Filedeskriptoren in    */
    {                                  /* Gebrauch; den aeltesten     */
        oldest = -1;                   /* suchen                      */
        for (i = 0; i < MAXOPEN; i++)
            if (oldest == -1 || fifos[i].time < oldest)
            {
                oldest = fifos[i].time;
                avail = i;
            }
        if (close(fifos[avail].fd) == -1)
            return(-1);
    }
```

```
    fifo = fifoname(key);
    if (mkfifo(fifo) == -1 && errno != EEXIST)
        return(-1);
                                        /* auf Sender warten          */
    for (tries = 1; tries <= MAXTRIES; tries++)
    {
        if ((fd = open(fifo, flags | O_NDELAY)) != -1)
            break;
        if (errno != ENXIO)
            return(-1);
        sleep(NAPTIME);
    }
    if (fd == -1)
    {
        errno = ENXIO;                  /* sleep kann es in Unordnung */
        return(-1);                     /* gebracht haben             */
    }
                                        /* O_NDELAY loeschen          */
    if (fcntl(fd, F_SETFL, flags) == -1)
        return(-1);
    fifos[avail].key = key;
    fifos[avail].fd = fd;
    fifos[avail].time = clock;
    return(fd);
}

BOOLEAN send(dstkey, buf, nbytes)       /* Message schicken           */
    long dstkey;
    char * buf;
    int nbytes;
{
    int fd;

    if ((fd = openfifo(dstkey, O_WRONLY)) == -1)
        return(FALSE);
    return(write(fd, buf, nbytes) != -1);
}
```

```
BOOLEAN receive(srckey, buf, nbytes)  /* Message empfangen          */
    long srckey;
    char * buf;
    int nbytes;
{
    int fd, nread;

    if ((fd = openfifo(srckey, O_RDONLY)) == -1)
        return(FALSE);
    while ((nread = read(fd, buf, nbytes)) == 0)
        sleep(NAPTIME);
    return(nread != -1);
}

void rmqueue(key)                     /* Message-Queue loeschen     */
    long key;
{
    int errno;

    if (unlink(fifoname(key)) == -1 && errno != ENOENT)
        syserr("unlink");
}
```

In Abschnitt 3.8 zeigten wir den Programmtext von **mkfifo**.

Um diese Funktionen bei der Arbeit zu sehen, zeigen wir hier ein einfaches Beispiel, das aus zwei Programmen besteht: ein Programm sendet Messages, und das andere empfängt sie. Die Struktur der Message ist in einer Definitionsdatei (**message.h**) festgelegt, so daß sie von beiden Programmen benutzt werden kann.

```
typedef struct
{
    long unused;
    int pid;
    int number;
} MESSAGE;
```

Die erste Komponente ist die geforderte unbenutzte long-Integer, die früher in diesem Abschnitt erklärt worden ist. Die zweite Komponente, **pid**, ist die Prozeßnummer des Senders. Der Empfänger kann diese als Schlüssel für die Queue benutzen, an die er Antworten zurück zum Sender schickt, obwohl dies unser erstes Beispiel nicht macht. Die letzte Komponente, **number**, ist die Message selbst, die in diesem Beispiel einfach eine Zahl ist.

Hier ist der Sender:

```
#include "message.h"

main()  /* Sender */
{
    MESSAGE m;

    m.pid = getpid();
    for (m.number = 1; m.number <= 4; m.number++)
    {
        sleep(1);
        if (!send(1000L, &m, sizeof(m)))
            syserr("send");
    }
    exit(0);
}
```

Und hier ist der Empfänger:

```
#include "message.h"

main()  /* Empfaenger */
{
    MESSAGE m;

    setbuf(stdout, NULL);
    while (receive(1000L, &m, sizeof(m)))
        printf("Received %d from %d\n", m.number, m.pid);
    syserr("receive");
}
```

Beachten Sie, daß der Schlüssel der Queue auf 1000 festgelegt ist — im allgemeinen keine gute Idee, im Moment aber ausreichend.

Als wir diese Kommandozeile tippten:

```
receiver & sender & sender & sender &
```

erhielten wir folgende Ausgabe:

```
Received 1 from 3723
Received 1 from 3724
Received 2 from 3724
Received 2 from 3723
Received 3 from 3723
```

```
Received 3 from 3724
Received 1 from 3725
Received 4 from 3723
Received 4 from 3724
Received 2 from 3725
Received 3 from 3725
Received 4 from 3725
```

Nun zu einem geringfügig komplizierteren Beispiel, dieses Mal mit einer Zwei-Weg-Kommunikation. Unsere zentralisierte Dienstleistung ist ein Prozeß, der zwei Zahlen addieren und das Resultat zurückschicken kann. Hier ist die Struktur der Message, die in der Definitionsdatei **addmsg.h** steht:

```
typedef struct
{
    long unused;
    long clientkey;
    int x;
    int y;
    int sum;
} MESSAGE;

#define ADDERKEY 1000L
```

Die Komponente **clientkey** wird vom zentralen Addierer zum Zurückschicken von Antworten an seine Klienten benützt. **x** und **y** sind die zu addierenden Zahlen, und **sum** ist deren Summe. Anstatt verschiedene Messages für die Inanspruchnahme des Dienstes und des Resultats zu verwenden, haben wir alle drei Komponenten in einer Struktur zusammengefaßt, so daß wir diese in beiden Kommunikationsrichtungen benützen können. Außerdem enthält die Definitionsdatei eine symbolische Konstante für den Schlüssel der Queue des addierenden Prozesses.

Hier ist das Programm des Addierers, der einfach eine Anfrage erhält, **x** und **y** addiert und die Summe zurückschickt:

```
#include "addmsg.h"

main()  /* Addierer */
{
    MESSAGE m;
    extern int errno;
```

```
    while (receive(ADDERKEY, &m, sizeof(m)))
    {
        m.sum = m.x + m.y;
        if (!send(m.clientkey, &m, sizeof(m)))
            printf("can't send to %ld; errno = %d\n",
                m.clientkey, errno);
    }
    syserr("receive");
}
```

Nun kommt das Programm eines Klienten, das eine Anfrage an den Addierer schickt, als Anwort eine Message erhält und das Resultat bezüglich der Arithmetik prüft:

```
#include "addmsg.h"

main()  /* Klient des Addierers */
{
    MESSAGE m;
    int x, y;

    m.clientkey = getpid();
    for (x = 1; x <= 5; x++)
        for (y = 1; y <= 5; y++)
        {
            m.x = x;
            m.y = y;
            if (!send(ADDERKEY, &m, sizeof(m)))
                syserr("send");
            if (!receive(m.clientkey, &m, sizeof(m)))
                syserr("receive");
            if (x + y != m.sum)
            {
                printf("Addition error!\n");
                exit(1);
            }
        }
    rmqueue(m.clientkey);
    printf("%ld worked OK\n", m.clientkey);
    exit(0);
}
```

Mit der Kommandozeile:

```
adder & addclient & addclient &
```

erhielten wir diese Ausgabe:

```
3757 worked OK
3758 worked OK
```

Da wir jetzt zwei einfache Beispiele gesehen haben, sind wir soweit, ein Mehrbenutzer-DBMS zu präsentieren. Da wir nicht den Platz für ein gutes DBMS haben, müssen wir uns unserem Schicksal fügen und uns mit einem schlechten zufriedengeben. Der springende Punkt ist jedoch, daß die Prozeßkommunikation selbst für ein gutes DBMS gut genug ist.

Unser Beispiel läßt sich in drei Teile aufspalten:

1. Das eigentliche DBMS, auf das über direkte Funktionsaufrufe aus dem eigenen Adreßraum zugegriffen wird. Das ist der Teil, der nicht so gut ist.
2. Der Mechanismus zur Prozeßkommunikation, der aus folgenden Teilen besteht: dem Programmtext auf der DBMS-Seite, der über Funktionsaufrufe auf Teil 1 zugreift, den Message-Werkzeugen selbst (**send**, **receive** und **rmqueue**) und einer Bücherei von Funktionen auf Seite des Klienten, die Messages zusammensetzen und Antworten empfangen.
3. Das Anwendungsprogramm, das ein Klient des DBMS ist. Es ruft Funktionen der Bücherei auf, die Teil 2 enthält, die dann im Auftrag mit dem DBMS kommunizieren.

Wir wollen mit Teil 1, dem eigentlichen DBMS, starten. Das DBMS kann einen Datensatz bestimmter Form in einer Datei speichern. Jeder Datensatz ist vom Typ **RCD**:

```
typedef struct
{
    char name[20];
    char street[15];
    char city[10];
    char state[3];
    char zip[6];
    char tel[15];
} RCD;
```

Die Datei ist eine Folge von Objekten des Typs **RCD**. Ein neuer Datensatz wird einfach am Ende der Datei angefügt — wir versuchen nicht, die Datensätze sortiert zu speichern. Ein bestehender Datensatz wird durch Überschreiben verändert und durch Zuweisen einer leeren Zeichenkette an die Komponente **name** gelöscht.

Alle Funktionen der Datenbank liefern einen Wert vom Typ **STATUS**:

```
typedef enum {OK, NOTFOUND, ERROR} STATUS;
```

Der Resultatwert **NOTFOUND** wird von einigen Funktionen nicht benützt, so daß deren Resultat in Wirklichkeit ein Boole'scher Wert ist. Aus Gründen der Einheitlichkeit benutzen wir jedoch trotzdem **STATUS**. Die Typvereinbarungen für **RCD** und **STATUS** stehen in der Definitionsdatei **dbms.h**.

Mittels **Dcreate** wird eine neue Datenbank angelegt, mittels **Dopen** eine bestehende geöffnet. Da immer nur eine Datenbank gleichzeitig geöffnet sein kann, schreiben diese Funktionen einfach den Filedeskriptor in eine globale Variable (**fd**). **Dclose** schließt eine Datenbank. Hier sind alle drei Funktionen:

```
#include "dbms.h"

static int fd = -1;

static STATUS Dopen(file)     /* Datenbank oeffnen     */
    char * file;
{
    if ((fd = open(file, O_RDWR, 0)) == -1)
        return(ERROR);
    return(OK);
}

static STATUS Dcreate(file)  /* Datenbank anlegen     */
    char * file;
{
    if ((fd = open(file, O_RDWR | O_CREAT | O_TRUNC, 0666)) == -1)
        return(ERROR);
    return(OK);
}

static STATUS Dclose()        /* Datenbank schliessen */
{
    if (close(fd) == -1)
        return(ERROR);
    return(OK);
}
```

Ein bestimmter Datensatz wird mittels **Dget** wiedergefunden, wobei folgende Argumente übergeben werden: der Wert der Komponente **name** und ein Zeiger auf eine **RCD**-Struktur, die den gefundenen Datensatz aufnehmen soll.

```
long lseek();
extern int errno;

static STATUS Dtop()          /* an den Anfang der Datenbank gehen */
{
    if (lseek(fd, 0L, 0) == -1)
        return(ERROR);
    return(OK);
}

static STATUS Dget(name, r)  /* Datensatz ueber Namen finden      */
    char * name;
    RCD * r;
{
    int nread;

    if (Dtop() != OK)
        return(ERROR);
    while ((nread = read(fd, r, sizeof(RCD))) == sizeof(RCD))
        if (strcmp(r->name, name) == 0)
        {                     /* einen Datensatz rueckwaerts gehen */
            if (lseek(fd, -(long)sizeof(RCD), 1) == -1)
                return(ERROR);
            return(OK);
        }
    switch (nread)
    {
        case 0:
            return(NOTFOUND);
        case -1:
            return(ERROR);
        default:
            errno = 0;
            return(ERROR);
    }
}
```

Dtop wird aufgerufen, um an den Anfang der Datei zu gelangen, da **Dget** sequentiell sucht, um den geforderten Datensatz zu finden. Nachdem ein Datensatz gefunden wurde, geht **Dget** an den Anfang dieses Datensatzes zurück, so daß dieser, falls er geändert werden soll, überschrieben werden kann.

Um Listen zu generieren, kann es sein, daß ein Klient alle Datensätze finden will. Hierzu wird erst **Dtop** aufgerufen, um an den Anfang der Datei zu positionieren, danach liefert **Dgetnext** bei jedem Aufruf den nächsten Datensatz. Falls **Dgetnext** das Ende der Datei erreicht, so liefert es **NOTFOUND**.

```
static STATUS Dgetnext(r)  /* naechsten Datensatz finden */
    RCD * r;
{
    while (1)
        switch (read(fd, r, sizeof(RCD)))
        {
            case sizeof(RCD):
                if (r->name[0] == '\0')
                    continue;
                return(OK);
            case 0:
                return(NOTFOUND);
            case -1:
                return(ERROR);
            default:
                errno = 0;
                return(ERROR);
        }
}
```

Mittels **Dput** wird ein Datensatz eingefügt. **Dput** ruft zuerst **Dget** auf, um zu sehen, ob ein Datensatz mit diesem Namen bereits existiert. Wenn dem so ist, wird dieser überschrieben; falls nicht, wird der neue Datensatz am Dateiende angefügt.

```
static STATUS Dput(r)                    /* Datensatz einfuegen */
    RCD * r;
{
    RCD rcd;

    switch (Dget(r->name, &rcd))
    {
        case NOTFOUND:
            if (lseek(fd, 0L, 2) == -1)  /* ans Dateiende gehen */
                return(ERROR);
            break;
        case ERROR:
            return(ERROR);
    }
```

```
    switch (write(fd, r, sizeof(RCD)))
    {
        case sizeof(RCD):
            return(OK);
        case -1:
            return(ERROR);
        default:
            errno = 0;
            return(ERROR);
    }
}
```

Schließlich löscht **Ddelete** einen Datensatz, indem dieser durch einen Datensatz mit leerem Namen überschrieben wird (sofern er überhaupt existiert). Wir unternehmen nichts, um den alten Platz wieder zu verwenden.

```
static STATUS Ddelete(name)  /* Datensatz loeschen */
    char * name;
{
    RCD rcd;

    switch (Dget(name, &rcd))
    {
        case NOTFOUND:
            return(OK);
        case ERROR:
            return(ERROR);
    }
    rcd.name[0] = '\0';
    switch (write(fd, &rcd, sizeof(RCD)))
    {
        case sizeof(RCD):
            return(OK);
        case -1:
            return(ERROR);
        default:
            errno = 0;
            return(ERROR);
    }
}
```

Beachten Sie, daß beim Löschen eines nichtexistierenden Datensatzes **OK** zurückgeliefert wird.

Das wärs — sicherlich eines der kompaktesten Datenbank-Managementsysteme der Welt! Es mangelt ihm nur an Flexibilität, Robustheit und Effizienz.

Wir könnten diese Zugriffsroutinen auf die Datenbank auch direkt benutzen, so wie in der dezentralisierten Methode in Abschnitt 7.2 beschrieben. Wir wollen sie jedoch statt dessen in einem zentralisierten DBMS benutzen und über Messages kommunizieren. Dafür müssen wir ein Hauptprogramm hinzufügen, das von einem Klienten eine Message empfängt, diese durch Aufruf der passenden Zugriffsfunktionen bearbeitet und dann das Resultat zurückschickt. Es folgt die Struktur für Messages, die wir benutzen wollen (sie ist in **dbms.h** vereinbart):

```
typedef struct
{
    long mtype;
    long clientkey;
    char cmd;
    char file[20];
    RCD rcd;
    STATUS status;
    int errno;
} MESSAGE;
```

Die ersten zwei Komponenten sind die gleichen wie bei dem Beispiel des Addierers. **cmd** ist ein einzelner Buchstabe, der das geforderte Kommando anzeigt. Falls ein Dateiname gefordert ist, wird dieser in **file** gespeichert. **rcd** wird sowohl zum Senden eines Datensatzes an das DBMS wie auch zum Empfang eines solchen benutzt. Der Status jeder Anfrage wird über **status** zurückgeliefert, und dementsprechend ist auch **errno** gesetzt (**errno** hat nur eine Bedeutung, falls **status** den Wert **ERROR** besitzt). Wiederum haben wir die gleiche Struktur für Anfragen und Resultate benützt, obwohl so manchmal einige Felder unbenutzt sind.

Hier ist nun das Hauptprogramm für den DBMS-Prozeß:

```
main()  /* Anfragen empfangen und Antworten schicken */
{
    MESSAGE m;
    char * dbmsval, name[30], * getenv();
    long dbmskey, atol();

    if ((dbmsval = getenv("DBMSKEY")) == NULL)
        fatal("missing DBMSKEY environment variable");
    dbmskey = atol(dbmsval);
```

```
    while (receive(dbmskey, &m, sizeof(m)))
    {
        switch (m.cmd)
        {
            case 'o':
                m.status = Dopen(m.file);
                break;
            case 'c':
                m.status = Dcreate(m.file);
                break;
            case 'q':
                m.status = Dclose();
                rmqueue(dbmskey);
                break;
            case 'g':
                strcpy(name, m.rcd.name);
                m.status = Dget(name, &m.rcd);
                break;
            case 'n':
                m.status = Dgetnext(&m.rcd);
                break;
            case 'p':
                m.status = Dput(&m.rcd);
                break;
            case 'd':
                m.status = Ddelete(m.rcd.name);
                break;
            case 't':
                m.status = Dtop();
                break;
            default:
                errno = EINVAL;
                m.status = ERROR;
        }
        m.errno = errno;
        if (!send(m.clientkey, &m, sizeof(m)))
            printf("can't send to %ld; errno = %d\n", m.clientkey, errno);
        if (m.cmd == 'q')
            exit(0);
    }
    syserr("receive");
}
```

Der Schlüssel der empfangenden Queue des DBMS wird über die Environment-Variable **DBMSKEY** ermittelt, die von der Shell vorbelegt wurde.

Ein Klient schickt Messages nicht direkt an das DBMS. Statt dessen simuliert eine Bücherei (voller Funktionen) die direkten Funktionen des DBMS, die oben beschrieben wurden. Tatsächlich muß ein Anwendungsprogrammierer noch nicht einmal wissen, ob das DBMS zentralisiert ist; er benutzt die gleichen Funktionen, die man auch bei einem dezentralisierten DBMS verwenden würde. Hier ist die DBMS-Bücherei:

```
#include "dbms.h"

extern int errno;
static MESSAGE m;

static STATUS dbmscall(r)  /* "Aufruf" des DBMS */
    RCD * r;
{
    char * dbmsval, * getenv();
    static long dbmskey;
    long atol();

    if (dbmskey == 0)
    {
        if ((dbmsval = getenv("DBMSKEY")) == NULL)
            fatal("missing DBMSKEY environment variable");
        dbmskey = atol(dbmsval);
    }
    if (m.clientkey == 0)
        m.clientkey = getpid();
    if (!send(dbmskey, &m, sizeof(m)))
        return(ERROR);
    if (!receive(m.clientkey, &m, sizeof(m)))
        return(ERROR);
    if (r != NULL)
        * r = m.rcd;
    if (m.status == ERROR)
        errno = m.errno;
    else
        errno = 0;
    return(m.status);
}
```

```
STATUS Dopen(file)          /* Datenbank oeffnen    */
    char * file;
{
    m.cmd = 'o';
    strcpy(m.file, file);
    return(dbmscall(NULL));
}

STATUS Dcreate(file)        /* Datenbank erzeugen   */
    char * file;
{
    m.cmd = 'c';
    strcpy(m.file, file);
    return(dbmscall(NULL));
}

STATUS Dclose()             /* Datenbank schliessen */
{
    STATUS status;

    m.cmd = 'q';
    status = dbmscall(NULL);
    rmqueue(m.clientkey);
    return(status);
}

STATUS Dtop()               /* an den Anfang der Datenbank gehen  */
{
    m.cmd = 't';
    return(dbmscall(NULL));
}

STATUS Dget(name, r)        /* ueber Namen einen Datensatz finden */
    char * name;
    RCD * r;
{
    m.cmd = 'g';
    strcpy(m.rcd.name, name);
    return(dbmscall(r));
}
```

```
STATUS Dgetnext(r)          /* naechsten Datensatz finden */
    RCD * r;
{
    m.cmd = 'n';
    return(dbmscall(r));
}

STATUS Dput(r)              /* Datensatz einfuegen        */
    RCD * r;
{
    m.cmd = 'p';
    m.rcd = * r;
    return(dbmscall(NULL));
}

STATUS Ddelete(name)        /* Datensatz loeschen         */
    char * name;
{
    m.cmd = 'd';
    strcpy(m.rcd.name, name);
    return(dbmscall(NULL));
}
```

Die Funktion **dbmscall** stellt das auf Messages basierende Gegenstück zum normalen Aufruf-Rücksprung-Muster eines Unterprogramms dar. **send** und **receive** verschleiern nicht nur die Tatsache, daß wir FIFO-Dateien benutzen, auch der Einsatz von Messages wird durch **dbmscall** versteckt. Dies ist wünschenswert, da eine versteckte Entwurfsentscheidung später leicht geändert werden kann. Falls wir uns entschließen, FIFO-Dateien durch Messages zu ersetzen (was wir in Kürze tun werden), können wir dies verwirklichen, indem wir einfach **send**, **receive** und **rmqueue** ändern, da der Rest unseres Programms nie wußte, daß wir zunächst einmal FIFO-Dateien benutzt haben. In ähnlicher Weise können wir auch entscheiden, die dezentralisierte Methode durchzuführen — dazu ändern wir einfach die DBMS-Bücherei. Das funktioniert nur, weil unsere Anwendungsprogramme sich der Tatsache der Zentralisierung nicht bewußt sind.

Zu guter Letzt benötigen wir ein Anwendungsprogramm, das als Testklient agiert. Es folgt ein interaktives Hilfsprogramm, das es einem Benutzer erlaubt, jede Funktion des DBMS auszuprobieren:

```
#include "dbms.h"

main()  /* interaktives Hilfsprogramm fuer das DBMS */
{
    char cmd[5], file[50], name[30];
    RCD rcd;
    extern int errno;

    while (1)
    {
        prompt("Command (? for help)", cmd, sizeof(cmd), TRUE);
        if (strlen(cmd) != 1)
        {
            printf("One letter only\n");
            continue;
        }
        switch (cmd[0])
        {
            case '?':
                printf("o   open database\n");
                printf("c   create database\n");
                printf("p   put record\n");
                printf("d   delete record\n");
                printf("g   get record by key\n");
                printf("n   get next record\n");
                printf("t   rewind to top of database\n");
                printf("q   quit\n");
                continue;
            case 'o':
                prompt("File to open", file, sizeof(file), TRUE);
                if (Dopen(file) == OK)
                    printf("OK\n");
                else
                    printf("FAILED; errno = %d\n", errno);
                continue;
            case 'c':
                prompt("File to create", file, sizeof(file), TRUE);
                if (Dcreate(file) == OK)
                    printf("OK\n");
                else
                    printf("FAILED; errno = %d\n", errno);
                continue;
```

```
case 'q':
    if (Dclose() == OK)
        printf("OK\n");
    else
        printf("FAILED; errno = %d\n", errno);
    exit(0);
case 'g':
    prompt("Name", name, sizeof(name), TRUE);
    switch (Dget(name, &rcd))
    {
        case OK:
            rcdprint(&rcd);
            continue;
        case NOTFOUND:
            printf("NOT FOUND\n");
            continue;
        case ERROR:
            printf("FAILED; errno = %d\n", errno);
            continue;
    }
case 'n':
    switch (Dgetnext(&rcd))
    {
        case OK:
            rcdprint(&rcd);
            continue;
        case NOTFOUND:
            printf("NOT FOUND\n");
            continue;
        case ERROR:
            printf("FAILED; errno = %d\n", errno);
            continue;
    }
    continue;
case 'p':
    prompt("Name", rcd.name, sizeof(rcd.name), TRUE);
    prompt("Street", rcd.street, sizeof(rcd.street), FALSE);
    prompt("City", rcd.city, sizeof(rcd.city), FALSE);
    prompt("State", rcd.state, sizeof(rcd.state), FALSE);
    prompt("ZIP", rcd.zip, sizeof(rcd.zip), FALSE);
    prompt("Tel", rcd.tel, sizeof(rcd.tel), FALSE);
```

```
                if (Dput(&rcd) == OK)
                    printf("OK\n");
                else
                    printf("FAILED; errno = %d\n", errno);
                continue;
            case 'd':
                prompt("Name", name, sizeof(name), TRUE);
                if (Ddelete(name) == OK)
                    printf("OK\n");
                else
                    printf("FAILED; errno = %d\n", errno);
                continue;
            case 't':
                if (Dtop() == OK)
                    printf("OK\n");
                else
                    printf("FAILED; errno = %d\n", errno);
                continue;
            default:
                printf("Unknown command - use ? for help\n");
        }
    }
}

static void prompt(msg, result, max, required)  /* Aufforderung    */
    char * msg, * result;                       /* an den Benutzer */
{
    char s[200];
    int len;

    while (1)
    {
        printf("\n%s? ", msg);
        if (gets(s) == NULL)
            exit(0);
        len = strlen(s);
        if (len >= max)
        {
            printf("Response too long\n");
            continue;
        }
```

```
        if (len == 0 && required)
        {
            printf("Value required\n");
            continue;
        }
        strcpy(result, s);
        return;
    }
}

static void rcdprint(r)  /* Datensatz ausgeben */
    RCD * r;
{
    printf("Name\t%s\n", r->name);
    printf("Street\t%s\n", r->street);
    printf("City\t%s\n", r->city);
    printf("State\t%s\n", r->state);
    printf("ZIP\t%s\n", r->zip);
    printf("Tel\t%s\n", r->tel);
}
```

Dem Programm ist ziemlich leicht zu folgen, insbesondere deshalb, weil die DBMS-Bücherei viele Details versteckt. Als Experiment können Sie versuchen, ob Sie irgend etwas finden, das auf die Kommunikation mit einem zentralisierten DBMS über Messages hinweist, anstatt auf Kommunikation mit einem dezentralisierten DBMS über Funktionsaufrufe. Falls Sie nichts finden können, bedeutet das folgendes: Hat man einmal die Prozeßkommunikation erledigt, so ist die zentralisierte Methode genauso leicht durch einen Anwendungsprogrammierer benutzbar wie die vertrautere dezentralisierte Methode.
Es folgt ein Beispiel einer Sitzung, um unser Testprogramm in Aktion zu zeigen. Bevor wir das Hilfsprogramm (**dbmstest**) starten, müssen wir **DBMSKEY** einen Wert zuweisen und das DBMS (das Programm **dbms**) im Hintergrund ausführen.

```
$ DBMSKEY=5000
$ dbms &
3789
$ dbmstest
```

```
Command (? for help)? ?
o   open database
c   create database
p   put record
d   delete record
g   get record by key
n   get next record
t   rewind to top of database
q   quit

Command (? for help)? c

File to create? tstdb
OK
Command (? for help)? p

Name? Mary Smith

Street? 123 Main St

City? Cityville

State? MD

ZIP? 20964

Tel? 301 345-9876
OK

Command (? for help)? p

Name? John Jones

Street? 222 Maple Dr

City? Hometown

State? OK
```

```
ZIP? 73112

Tel? 405 878-1445
OK

Command (? for help)? g

Name? Mary Smith
Name    Mary Smith
Street  123 Main St
City    Cityville
State   MD
ZIP     20964
Tel     301 345-9876

Command (? for help)? t
OK

Command (? for help)? n
Name    Mary Smith
Street  123 Main St
City    Cityville
State   MD
ZIP     20964
Tel     301 345-9876

Command (? for help)? n
Name    John Jones
Street  222 Maple Dr
City    Hometown
State   OK
ZIP     73112
Tel     405 878-1445

Command (? for help)? q
OK
$
```

7.5 Systemaufrufe zu Messages (System V)

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int msgget(key, flags)  /* Schluessel der Message-Queue holen     */
    key_t key;          /* Schluessel der Queue (long)            */
    int flags;          /* Flagge fuer Optionen                   */

/* liefert Schluesselnummer oder -1 bei Fehler                    */


int msgsnd(qid, buf, nbytes, flags)  /* Message an Queue schicken */
    int qid;                         /* Nummer der Queue         */
    struct msgbuf * buf;             /* Zeiger auf Message       */
    int nbytes;                      /* Groesse der Message      */
    int flags;

/* liefert 0 bei Erfolg oder -1 bei Fehler                        */


int msgrcv(qid, buf, nbytes, mtype, flags)  /* Message empfangen  */
    int qid;
    struct msgbuf * buf;
    int nbytes;
    long mtype;                          /* Art der Message       */
    int flags;

/* liefert Anzahl empfangener Bytes oder -1 bei Fehler            */


int msgctl(qid, cmd, sbuf)   /* Message-Queue manipulieren        */
    int qid;
    int cmd;                 /* Kommando                          */
    struct msqid_ds * sbuf;  /* Zeiger auf Statuspuffer           */

/* liefert 0 bei Erfolg oder -1 bei Fehler                        */
```

Diese Systemaufrufe, die nur in System V vorhanden sind, behandeln Messages analog zu **send** und **receive**, die wir im vorigen Abschnitt mit FIFO-Dateien implementierten. Das Spektrum in System V ist jedoch wesentlich reichhaltiger und es

gibt viel mehr Optionen, als irgendwer wahrscheinlich jemals benötigt. Man kann sich beinahe einen Ausschuß vorstellen, der diese Systemaufrufe entwirft und Angst hat, zu vertagen, bevor nicht jede denkbare Möglichkeit vorhanden ist, damit ihn niemand wegen zu starker Vereinfachung anklagen kann.

Wir werden das meiste erklären, was diese Systemaufrufe tun. Die etwas ausgefalleneren Möglichkeiten überspringen wir allerdings und konzentrieren uns auf die wesentlichen Punkte, die es uns ermöglichen, **send** und **receive** nachzubauen.

Um Messages zu benutzen, beginnt man mit **msgget**, das ähnlich zu **open** ist. **msgget** erwartet als Argument einen Schlüssel, der eine long-Integer sein muß, und liefert als Resultat eine Integer, die *Schlüsselnummer* (*queue-ID*) genannt wird. Eine Schlüsselnummer ist wie ein Filedeskriptor, mit der Ausnahme, daß jeder Prozeß, der ihren Wert kennt, sie benützen kann — er muß nicht vererbt werden, um gültig zu sein. Eine Schlüsselnummer ist, wie ein Filedeskriptor auch, ein Index in eine Tabelle im Kern. Indem man externe Schlüssel in Schlüsselnummern übersetzt, kann zeitintensives Aufsuchen vermieden werden, wenn auf die Queue verwiesen werden muß.

Das Argument **flags** für **msgget** verhält sich ähnlich wie eine Kombination des zweiten und dritten Arguments bei **open**. Wenn das **IPC_CREAT**-Bit gesetzt ist, wird die Queue erzeugt, falls sie noch nicht existiert. In diesem Fall bilden die neun am weitesten rechts stehenden Bits von **flags** den Zugriffsschutz der Queue. Die Verwendung dieser Bits ist wieder fast wie bei Dateien: Schreibrecht benötigt man, um eine Message zu schicken, und Leserecht, um eine zu empfangen. Ist das **IPC_CREAT**-Bit nicht gesetzt, so muß die Queue bereits existieren. In diesem Fall wird **msgget** nur zum Übersetzen eines Schlüssels in eine Schlüsselnummer benützt.

Falls das Argument **key** gleich **IPC_PRIVATE** ist, so wird eine Queue erzeugt, egal ob in **flags** das **IPC_CREAT**-Bit gesetzt ist oder nicht. Dies vermeidet einen Konflikt mit einer bestehenden Queue, der bei Verwendung eines wirklichen Schlüssels möglich wäre. Auch ohne diese Möglichkeit könnte man einen Algorithmus zum Erzeugen einer privaten Queue programmieren, indem man ähnliche Techniken wie beim Erzeugen temporärer Dateien mit eindeutigen Namen verwendet. In unseren Beispielen werden wir die Prozeßnummer als Schlüssel für eine private Queue benutzen.

Hat man einmal eine Schlüsselnummer, so kann man **msgsnd** aufrufen, um eine Message in diese Queue zu stecken. Das zweite Argument **buf** muß auf eine willkürliche Struktur zeigen, die mit einer positiven long-Integer beginnen muß, dem sogenannten *Message-Typ*. Alle Objekte vom Typ **MESSAGE** in den Definitionsdateien des vorigen Abschnitts genügen dieser Bedingung. Message-Typen erlauben dem Empfänger, eine Message der Queue nach Typ auszuwählen oder auf eine Message eines bestimmten Typs zu warten. **nbytes** ist die Anzahl Bytes in der Message (ohne die Bytes für den Message-Typ). Das letzte Argument, **flags**, ist im Normalfall 0 und veranlaßt **msgsnd** zum Blockieren, falls die Queue voll ist. Ist es statt dessen auf **IPC_NOWAIT** gesetzt, so kehrt **msgsnd** bei einer vollen Queue sofort mit

einem Fehler zurück. Diese Flagge verhält sich fast genauso wie **O_NDELAY** bei FIFO-Dateien. (Sie hätte wahrscheinlich **IPC_NDELAY** genannt werden sollen.)

Message-Typen vervielfachen eine einzelne Message-Queue optisch, damit diese eine Menge von untergeordneten Queues gleichzeitig behandeln kann, für jeden Typ eine. Falls die Anzahl der verschiedenen Typen nicht sehr groß ist — eine ungewöhnliche Situation — so könnten auch mehrere verschiedene Queues benützt werden, von denen jede nur Messages eines Typs behandeln würde.

Der Empfänger ruft **msgrcv** auf, wobei **nbytes** auf die maximale Größe einer Message (in Bytes) gesetzt wird, die in die Speicherfläche paßt, auf die **buf** zeigt. Auch hier wird die Komponente für den Typ wieder nicht mitgezählt. Da die Größe der tatsächlichen Message, die empfangen wird, kleiner als **nbytes** sein kann, wird gerade dieser Wert als Resultat von **msgrcv** geliefert (analog zu **read**). Möchte der Empfänger nur Messages eines bestimmten Typs, so setzt er **mtype** auf die entsprechende Typnummer. Anderenfalls sollte **mtype** Null sein, dann empfängt man die älteste Message der Queue, ungeachtet des Typs. Falls keine geeignete Message in der Queue ist und weiterhin **flags** gleich Null ist, so blockiert der Empfänger. Ist dagegen die **IPC_NOWAIT**-Flagge gesetzt, so kehrt der Empfänger, anstatt zu Blockieren, mit einem Fehler zurück.

msgctl befragt oder manipuliert verschiedene Eigenschaften der Queue: Zugriffsrechte, Eigentumsverhältnisse und Kapazität. Wir überspringen die Details; wichtig ist jedoch, daß das Kommando **IPC_RMID** eine Message-Queue zerstört. Jeder Prozeß, der zu diesem Zeitpunkt in **msgsnd** oder in **msgrcv** blockiert ist, wird ein Fehlerresultat empfangen.

Hier sind neue Fassungen von **send**, **receive** und **rmqueue**, bei der Messages aus System V benützt werden; Sie sollten diese Lösung vielleicht mit der Lösung aus dem vorigen Abschnitt vergleichen, die FIFO-Dateien verwendet. Alle Messages sind vom Typ 1, und ein Sender bzw. Empfänger blockiert, falls die Queue voll bzw. leer ist. Dies ist in Ordnung, da im Unterschied zu FIFO-Dateien der Sender Messages in die Queue schicken kann, ohne auf den Empfänger zu warten — es gibt hier kein Konzept des "Öffnens zum Lesen".

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

#define MAXOPEN 20
```

```
static int openqueue(key)              /* Schluesselnummer liefern */
    long key;                          /* falls noetig, erzeugen   */
{
    static struct
    {
        long key;
        int qid;
    } queues[MAXOPEN];
    int i, avail, qid;
    extern int errno;

    avail = -1;
    for (i = 0; i < MAXOPEN; i++)
    {
        if (queues[i].key == key)
            return(queues[i].qid);
        if (queues[i].key == 0 && avail == -1)
            avail = i;
    }
    if (avail == -1)
    {
        errno = 0;
        return(-1);
    }
    if ((qid = msgget(key, 0666 | IPC_CREAT)) == -1)
        return(-1);
    queues[avail].key = key;
    queues[avail].qid = qid;
    return(qid);
}

BOOLEAN send(dstkey, buf, nbytes)      /* Message schicken         */
    long dstkey;
    struct msgbuf * buf;
    int nbytes;
```

```
{
    int qid;

    if ((qid = openqueue(dstkey)) == -1)
        return(FALSE);
    buf->mtype = 1;
    return(msgsnd(qid, buf, nbytes - sizeof(buf->mtype), 0) != -1);
}

BOOLEAN receive(srckey, buf, nbytes)  /* Message empfangen */
    long srckey;
    struct msgbuf * buf;
    int nbytes;
{
    int qid;

    if ((qid = openqueue(srckey)) == -1)
        return(FALSE);
    return(msgrcv(qid, buf, nbytes - sizeof(buf->mtype), 0L, 0) != -1);
}

void rmqueue(key)                    /* Queue loeschen    */
    long key;
{
    int qid;

    if ((qid = openqueue(key)) == -1
         || msgctl(qid, IPC_RMID, NULL) == -1)
        syserr("rmqueue");
}
```

Beachten Sie, daß wir 20 Message-Queues erlauben und keine von diesen löschen, um Platz für neue zu schaffen, da sie als Betriebsmittel nicht so knapp sind wie Filedeskriptoren.

Warum sollte man nun Messages (aus System V) den FIFO-Dateien vorziehen? Messages sind zwar schneller, aber doch nicht so viel, um ihre Benutzung in einer Anwendung zu rechtfertigen, wenn nicht die Zeit für die Übergabe von Messages ein Problem darstellt. Messages bieten viele schicke Möglichkeiten (von denen sehr wenige von **send** und **receive** benützt werden), doch werden diese in den meisten Anwendungen nicht benötigt. Außerdem sind Messages kompliziert, unvollständig dokumentiert und nicht portabel. *Vermeiden Sie sie, wenn irgend möglich!*

7.6 Semaphore

7.6.1 Grundlagen zur Benutzung von Semaphoren

Ein Semaphor ist ein Mechanismus, der zwei oder mehr Prozesse daran hindert, auf ein exklusiv benötigtes, gemeinsames Betriebsmittel gleichzeitig zuzugreifen. Bei der Eisenbahn verhindert ein Signal einen Zusammenstoß zweier Züge auf einem gemeinsam benutzten Gleisabschnitt. Sowohl bei der Eisenbahn wie auch beim Computer haben Semaphore nur eine beratende Funktion: Wenn der Lokomotivführer das Signal nicht beachtet oder sich nicht danach richtet, so kann auch das Signal einen Zusammenstoß nicht verhindern. Gleiches gilt, wenn ein Prozeß ein Semaphor nicht prüft, bevor er auf ein gemeinsames Betriebsmittel zugreift — auch hier kann dann Chaos entstehen.

Ein *binäres* Semaphor hat nur zwei Zustände: blockiert (*locked*) und frei (*unlocked*). Ein *allgemeines* Semaphor hat unendlich viele (oder zumindest sehr viele) Zustände. Es ist ein Zähler, der bei Erwerb des Semaphors (*locked*) um eins erniedrigt und bei Freigabe (*unlocked*) um eins erhöht wird. Hat das Semaphor den Wert Null, so muß ein Prozeß, der es erwerben will, auf einen anderen Prozeß warten, der es hochzählt — es kann niemals negativ werden. (Kann ein Semaphor nur zwei Werte annehmen, so ist es ein binäres Semaphor.) Auf allgemeine Semaphore wird üblicherweise über zwei Funktionen namens **P** (Erwerb) und **V** (Freigabe)[4] zugegriffen.[5] Wir können diese in C etwa wie folgt formulieren:

```
void P(sem)  /* Semaphor erwerben  */
    int * sem;
{
    while (* sem <= 0)
        ;    /* nichts tun         */
    (* sem)--;
}

void V(sem)  /* Semaphor freigeben */
    int * sem;
{
    (* sem)++;
}
```

Das Semaphor muß durch Aufruf von **V** initialisiert werden; andernfalls startet das Semaphor ohne Erwerbsmöglichkeit. Falls das Semaphor zum Beispiel die Anzahl

[4] Protect und Vrej. A.d.Ü.

[5] P und V sind Abkürzungen für die holländischen Worte für "wait" und "signal". Wir benutzen statt dessen "Erwerb" und "Freigabe", da "wait" und "signal" in UNIX andere Bedeutungen haben. Außerdem spreche ich nicht Holländisch.

freier Puffer zählt, und am Anfang fünf davon existieren, so würden wir beginnen, indem wir **V** fünfmal aufrufen. Alternativ können wir auch einfach die Semaphor-Variable auf fünf setzen.

Nachdem wir uns die Mühe gemacht haben, **P** und **V** zu programmieren, müssen wir Ihnen nun mitteilen, daß diese Funktionen aus drei Gründen nicht funktionieren werden:

- Die Semaphor-Variable, auf die **sem** verweist, kann nicht von mehreren Prozessen gemeinsam benutzt werden, die unterschiedliche Datensegmente haben.
- Unsere Funktionen werden nicht atomar ausgeführt — der Kern kann einen Prozeß zu jedem beliebigen Zeitpunkt unterbrechen. Folgendes Szenario könnte sich ergeben: Prozeß 1 beendet die **while**-Schleife in **P** und wird unterbrochen, bevor er das Semaphor herunterzählen kann; Prozeß 2 beginnt **P**, findet das Semaphor bei 1, beendet seine **while**-Schleife und erniedrigt das Semaphor auf 0; Prozeß 1 macht weiter und verringert den Wert des Semaphors auf –1 (ein unzulässiger Wert).
- **P** führt ein sogenanntes *busy-wait* aus, falls **sem** den Wert Null besitzt. Dies ist keine intelligente Art, eine CPU zu benutzen.

Deshalb können **P** und **V** nicht einfach im Adreßraum des Benutzers programmiert werden. Semaphore müssen durch den Kern unterstützt werden, der Daten zwischen Prozessen teilen, atomare Operationen ausführen und die CPU einem rechenbereiten Prozeß zuteilen kann, falls ein anderer blockiert ist.

7.6.2 Implementierung von Semaphoren mittels Messages

In Abschnitt 2.5 zeigten wir, wie man **creat** benutzt, um ein primitives binäres Semaphor zu erzeugen. In Abschnitt 2.6 verbesserten wir diese Technik, indem wir die **O_EXCL**-Flagge bei **open** benützten. Solche Methoden sind ausreichend für einen Prozeß, der auf ein gemeinsames Betriebsmittel nur ein paarmal zugreift, wie zum Beispiel bei einem Programm zur Postabwicklung, das in eine Briefkasten-Datei schreibt. Für Hochleistungsanwendungen ist der Aufwand dazu jedoch viel zu hoch. Wir wollen ein Semaphor, das weniger Zeit zum Prüfen und Setzen benötigt.

Eine Message-Queue kann als Semaphor benutzt werden: **send** fügt eine Message zur Queue hinzu und ist äquivalent zu **V**; das Analogon zu **P** heißt **receive** und nimmt eine Message aus der Queue heraus. **receive** blockiert, falls die Queue leer ist, und verhält sich somit wie ein Semaphor mit Wert Null.

Hier sind **P** und **V**, beide mittels **send** und **receive** implementiert:

```
void P(sem)  /* Semaphor erwerben  */
    int sem;
{
    long type;

    if (!receive((long)sem, &type, sizeof(type)))
        syserr("receive");
}

void V(sem)  /* Semaphor freigeben */
    int sem;
{
    long type;

    if (!send((long)sem, &type, sizeof(type)))
        syserr("send");
}
```

Wir verwenden nun als Argument eine Integer-Variable (**sem**) und nicht mehr einen Zeiger auf eine Integer. Da die Message keinen Inhalt hat, haben wir ihr einfach eine long-Integer zugeordnet; dies genügt der Anforderung, daß ein Message-Typ am Anfang der Message steht. Beide Implementierungen von **send** und **receive**, FIFO-Dateien oder Messages aus System V, funktionieren auf jeden Fall.

7.6.3 Semaphore in System V

Messages für Semaphore zu benutzen ist bereits wesentlich schneller als Dateien zu erzeugen, jedoch entsteht durch das Hinzufügen bzw. Löschen einer Message an bzw. aus einer Queue doch noch mehr unnötiger Aufwand als durch einfaches Hoch- oder Herunterzählen einer Variablen. Deshalb besitzen sowohl System V wie auch Xenix 3 eine Reihe von Systemaufrufen speziell zur Behandlung von Semaphoren. Leider verwendet keines der beiden Systeme einfach **P** und **V** — ihre Semaphore sind reichhaltiger und umständlicher.

Wir wollen mit System V beginnen. Diese Systemaufrufe sind zu komplex für uns, um sie vollständig zu beschreiben.[6] Wir wollen allerdings sichergehen, daß wir genügend erklären, um uns eine Implementierung von **P** und **V** zu ermöglichen.

[6] Um ehrlich zu sein, muß ich zugeben, daß ich sie nicht vollständig *verstehe*.

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int semget(key, nsems, flags)     /* Nummer des Semaphor-Sets holen    */
    key_t key;                    /* Schluessel des Semaphor-Sets      */
    int nsems;                    /* Anzahl Semaphore                  */
    int flags;                    /* Optionen-Flaggen                  */

/* liefert Nummer des Semaphor-Sets oder -1 bei Fehler                 */


int semop(sid, ops, nops)         /* Operation auf Semaphor-Set        */
                                  /* ausfuehren                        */
    int sid;                      /* Nummer des Semaphor-Sets          */
    struct sembuf (* ops)[];      /* Zeiger auf Vektor von Operationen */
    int nops;                     /* Anzahl Operationen                */

/* liefert Wert des Semaphors vor der letzten Operation                */
/* oder -1 bei Fehler                                                  */


int semctl(sid, snum, cmd, arg)   /* Semaphor manipulieren             */
    int sid;
    int snum;                     /* Nummer des Semaphors              */
    int cmd;                      /* Kommando                          */
    char * arg;                   /* Argument                          */

/* liefert kommandoabhaengigen Wert oder -1 bei Fehler                 */
```

Diese Systemaufrufe scheinen von jemandem gemacht zu sein, der die Programmiersprache APL liebt: **semget** holt einen *Vektor* von Semaphoren, auf denen auch noch gleichzeitig (unteilbar) operiert wird, indem an **semop** ein Vektor von Operationen übergeben wird.

Semaphore in System V sind zu kompliziert. In jedem Programm, das Semaphore benutzt, ist es lebenswichtig, daß man beweisen kann, daß der Zugriff auf gemeinsame Betriebsmittel exklusiv stattfindet, und daß keine Deadlock- oder Hungertod-Situation (Zugriff wird niemals erlangt) auftritt. Da die Dinge sehr zeitabhängig sind, genügt Testen allein in der Regel nicht; folglich muß gründliche Analyse betrieben werden. Dies ist schon schwer genug mit reinen **P**- und **V**-Operationen — werden dagegen die Systemaufrufe von System V in ihrer vollen Pracht benützt, so ist eine Analyse wahrscheinlich unmöglich.

Wir zeigen nun kurz, wie diese Aufrufe arbeiten. **semget** übersetzt einen Schlüssel in eine Nummer, die eine Menge von Semaphoren repräsentiert. Falls in **flags** das **IPC_CREAT**-Bit gesetzt ist, so wird der Satz erzeugt, falls er noch nicht existiert. In der Menge gibt es **nsems** Semaphore, die ab Null numeriert werden. Es ist relativ sinnvoll, in einer Menge nur ein Semaphor zu besitzen, falls dies irgend möglich ist. Zur Vereinfachung nehmen wir an, daß das für den Rest dieser Beschreibung der Fall ist, das heißt also, daß eine Semaphor-Operation oder ein Kontrollkommando auf ein einzelnes Semaphor wirkt, und somit die Nummer eines Semaphor-Satzes auch als Nummer eines Semaphors bezeichnet werden kann.

Um ein Semaphor zu erwerben oder freizugeben, wird **semop** benützt. **ops** zeigt hierbei auf eine Struktur, die drei short-Integer enthält:

```
struct sembuf
{
    short sem_num;  /* Nummer des Semaphors        */
    short sem_op;   /* Operation                   */
    short sem_flg;  /* Optionen fuer die Operation */
};
```

Die Operation kann eine negative oder positive Integer sein, oder auch Null. Im Gegensatz zu einer **P**-Operation, die wartet, bis ein Semaphor um 1 verringert werden kann, kann man hier auch warten, bis das Semaphor sich um einen gewissen Betrag herunterzählen läßt. Auch dies ist wieder eine Möglichkeit, von der wir Abstand nehmen wollen, indem wir nur mit zwei Werten für **op** arbeiten, -1 und 1. Dies entspricht gerade den Operationen **P** und **V**. Falls **op** den Wert -1 besitzt, blockiert der Prozeß, bis das Semaphor um eins verringert werden kann, ohne negativ zu werden. Ist **op** dagegen 1, so wird das Semaphor um eins erhöht. Bei beiden Operationen kann **flags** den Wert **SEM_UNDO** besitzen, wodurch **P** und **V** im Gleichgewicht gehalten werden, falls der Prozeß endet.

semctl kann zur Ermittlung oder Änderung des Besitzers, der Zugriffsrechte oder des Zeitstempels der letzten Änderung eines Semaphors benutzt werden. Weiterhin kann man herausfinden, wieviele Prozesse gerade auf das Semaphor warten, und welcher Prozeß es das letzte Mal geändert hat.

Hier sind nun **P** und **V**, mit **semget** und mit **semop** implementiert. Wir haben einen Aufruf zum Übersetzen eines Schlüssels in eine Semaphor-Nummer hinzugefügt. Im allgemeinen wird **semtran** einmal zu Beginn jedes Prozesses aufgerufen, der Semaphore benützt.

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
```

```
int semtran(key)  /* Semaphor-Schluessel in Nummer uebersetzen */
    int key;
{
    int sid;

    if ((sid = semget((key_t)key, 1, 0666 | IPC_CREAT)) == -1)
        syserr("semget");
    return(sid);
}

static void semcall(sid, op)  /* semop aufrufen */
    int sid;
    int op;
{
    struct sembuf sb;

    sb.sem_num = 0;
    sb.sem_op = op;
    sb.sem_flg = 0;
    if (semop(sid, &sb, 1) == -1)
        syserr("semop");
}

void P(sid)  /* Semaphor erwerben  */
    int sid;
{
    semcall(sid, -1);
}

void V(sid)  /* Semaphor freigeben */
    int sid;
{
    semcall(sid, 1);
}
```

Diese Implementierung von **P** und **V** ist im Betrieb viel effizienter als diejenige mit FIFO-Dateien oder Messages.

Als wir **P** und **V** mittels **send** und **receive** implementierten, benötigten wir **semtran** nicht. Um Anwendungen, die Semaphore benutzen, portabler zu machen, können wir der dortigen Implementierung allerdings eine identische Funktion hinzufügen:

```
int semtran(key)  /* Semaphor-Schluessel in Nummer uebersetzen */
    int key;
{
    return(key);
}
```

7.6.4 Semaphore in Xenix 3

In Xenix 3 wird ein anderer Mechanismus für Semaphore benutzt. Dieser kann nur binäre Semaphore behandeln und ist wunderbar einfach.

```
int creatsem(path, perms)  /* Semaphor-Datei erzeugen              */
    char * path;           /* Pfadname                             */
    int perms;             /* Zugriffsrechte-Bits                  */

/* liefert Nummer des Semaphors oder -1 bei Fehler                 */


int opensem(path)          /* existierende Semaphor-Datei oeffnen */
    char * path;

/* liefert Nummer des Semaphors oder -1 bei Fehler                 */


int waitsem(sid)           /* Semaphor erwerben                    */
    int sid;               /* Nummer des Semaphors                 */

/* liefert 0 bei Erfolg oder -1 bei Fehler                         */


int nbwaitsem(sid)         /* Semaphor pruefen                     */
    int sid;

/* liefert 0 bei Erfolg oder -1 bei Fehler                         */


int sigsem(sid)            /* Semaphor freigeben                   */
    int sid;

/* liefert 0 bei Erfolg oder -1 bei Fehler                         */
```

In Xenix 3 ist ein Semaphor eine neue Art von Datei — mit einer Inode, aber ohne Daten. Die Dateiattribute der Datei haben Bit 050000 gesetzt, das glücklicherweise von System V und 4.2 BSD nicht benutzt wird. Nur das Leserecht wird benützt — besitzt ein Prozeß dieses, so kann er ein Semaphor öffnen, es erwerben und es freigeben.

creatsem und **opensem** funktionieren fast wie **creat** und **open**. Hat man einmal eine Nummer eines Semaphors, so kann man **waitsem** aufrufen, um es zu erwerben, oder **sigsem**, um es freizugeben (als Hinweis für einen Prozeß, der auf das Semaphor wartet). Warten mehrere Prozesse, so bekommt der älteste das Semaphor zuerst — eine *First-In-First-Out*-Queue liegt vor (nicht zu verwechseln mit einer FIFO-Datei). **nbwaitsem** ist eine nichtblockierende Form von **waitsem**. Ist das Semaphor blockiert, so kehrt **nbwaitsem** sofort zurück, und **errno** ist auf **ENAVAIL** gesetzt (ein neuer Fehler-Code, der in Xenix 3 hinzugefügt wurde).

Es ist sehr leicht, **semtran**, **P** und **V** bei Xenix 3 erneut zu implementieren:

```
int semtran(key)  /* Semaphor-Schluessel in Nummer uebersetzen */
    int key;
{
    int sid;
    char sname[20];
    extern int errno;

    sprintf(sname, "/tmp/sem%d", key);
    if ((sid = opensem(sname)) != -1)
        return(sid);
    if (errno == ENOENT)
        if ((sid = creatsem(sname, 0444)) != -1)
            return(sid);
    syserr("semtran");
}

void P(sid)  /* binaeres Semaphor erwerben  */
    int sid;
{
    if (waitsem(sid) == -1)
        syserr("waitsem");
}
```

```
void V(sid)  /* binaeres Semaphor freigeben */
    int sid;
{
    if (sigsem(sid) == -1)
        syserr("sigsem");
}
```

Vergessen Sie nicht, daß diese Implementierung nur binäre Semaphore unterstützt. Falls Sie **V** zweimal hintereinander aufrufen, bekommen Sie von **sigsem** ein Fehlerresultat. Für die Entwickler von Xenix 3 wäre es genauso einfach gewesen, allgemeine Semaphore zu unterstützen. In Anbetracht unserer harten Kritik an den System V Semaphoren sollten wir den Entwicklern bei Microsoft jedoch für ihre Zurückhaltung danken.

Wenn man zwei oder mehr Semaphore benützt, ist immer eine sogenannte Deadlock-Situation möglich. Weder System V noch Xenix 3 kontrollieren dies, und auch unsere Implementierung mittels FIFO-Dateien tut nichts dergleichen. Wenn möglich, sollte man seine Anwendung immer so entwerfen, daß sie keine Deadlock-Situationen enthält; je weniger Semaphore man benützt, desto leichter ist dies. Die beste Lösung — allerdings nicht immer gangbar — ist, überhaupt keine Semaphore zu verwenden. Bei einem zentralisierten DBMS zum Beispiel, das über Messages kommuniziert, sind die einzigen Semaphore innerhalb des DBMS die Satzsperren (*record locks*).

In diesem speziellen Fall ist das DBMS niemals an einer Deadlock-Situation beteiligt — nur seine Klienten. Deshalb kann das DBMS so programmiert werden, daß es die Deadlock-Situation erkennen und sie dann beseitigen kann: im Normalfall durch die Terminierung eines Klienten, der an der Deadlock-Situation beteiligt ist.

7.7 Shared-Memory

7.7.1 Grundlagen zur Benutzung von Shared-Memory

Die schnellste Art, Daten von einem Prozeß zu einem anderen zu übertragen, besteht darin, sie überhaupt nicht zu übertragen. Sender und Empfänger benützen den gleichen Speicherbereich, und wenn der Sender die Daten dort ablegt, so sind sie sofort für den Empfänger verfügbar. Um den Empfänger daran zu hindern, daß er die Daten zu früh liest, wird ein Semaphor oder eine Message benutzt. Genauso wird dadurch der Sender gehindert, neue Daten zu schicken, bevor der Leser die alten abgearbeitet hat.

Sowohl System V als auch Xenix 3 unterstützen beide *gemeinsame Speicherbereiche* (*Shared-Memory*) für beliebig viele Prozesse. Der gemeinsame Speicherbereich wird ein *Segment* genannt. Es kann mehrere gemeinsame Segmente geben, wobei

jedes für eine Untermenge der aktiven Prozesse verfügbar ist. Weiterhin kann ein Prozeß auf mehrere gemeinsame Segmente zugreifen.

In beiden Implementierungen wird das Segment zuerst außerhalb der Adreßräume aller Prozesse erzeugt. Danach führt jeder Prozeß, der auf das Segment zugreifen will, einen Systemaufruf aus, um es in seinen eigenen Adreßraum abzubilden. Auf modernen Computern werden mehrere Hardware-Segmentierungsregister benutzt, um Datensegmente zu adressieren. Indem man eines oder mehrere dieser Register freihält, geht das Abbilden auf ein gemeinsames Segment sehr schnell. Weitere Zugriffe auf den gemeinsamen Speicherbereich geschehen mit normalen Maschinenbefehlen, die Daten holen und speichern. Da das gemeinsame Segment innerhalb des Adreßraums des Prozesses liegt, erfolgt der Zugriff genauso schnell wie auf lokale Variablen.

Durch Hardware-Restriktionen kann die Anzahl der gemeinsamen Segmente, auf die ein Prozeß gleichzeitig zugreifen kann, beschränkt sein. Um Portabilität auf andere Computer zu gewährleisten, ist es deshalb das beste, immer nur ein Segment auf einmal in den eigenen Adreßraum abzubilden. Auch die Größe eines Segments kann durch die Hardware limitiert sein. Auf einer PDP-11 zum Beispiel kann ein Segmentierungsregister nur 8192 Bytes adressieren, was die maximale Größe eines Segments darstellt (dies ist für die meisten Anwendungen bei weitem ausreichend).

Zwischen den Implementierungen in System V und in Xenix 3 gibt es einen wesentlichen Unterschied. Bei System V kann ein Prozeß ein gemeinsames Segment in seinen Adreßraum abbilden und es dort längere Zeit behalten. Diese Methode ist effizient, da das Abbilden in den Adreßraum vielfach langsamer ist als der Zugriff auf das Segment mit direkten Maschinenbefehlen (obwohl es schneller ist als die meisten Systemaufrufe). Im Gegensatz dazu muß in Xenix 3 ein Prozeß jedesmal, bevor er das Segment benutzen will, es erst in seinen Adreßraum abbilden und danach wieder freigeben. Dies liegt daran, daß es verboten ist, Systemaufrufe auszuführen, während ein Segment im Adreßraum abgebildet ist. Abhängig von der Anwendung, kann in Xenix 3 Shared-Memory um vieles langsamer sein als in System V. Trotzdem ist Shared-Memory in beiden Systemen immer noch viel schneller als irgendeine andere Art der Prozeßkommunikation.

7.7.2 Shared-Memory in System V

Die Systemaufrufe für Shared-Memory in System V ähneln denen für Messages und Semaphore:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
```

```
int shmget(key, nbytes, flags)   /* Segmentnummer holen      */
    key_t key;                   /* Segmentschluessel        */
    int nbytes;                  /* Groesse des Segments     */
    int flags;                   /* Optionen-Flaggen         */

/* liefert Segmentnummer oder -1 bei Fehler                  */


char * shmat(segid, addr, flags) /* Segment abbilden         */
    int segid;                   /* Segmentnummer            */
    char * addr;                 /* gewuenschte Adresse      */
    int flags;

/* liefert Segmentadresse oder -1 bei Fehler                 */


int shmdt(addr)                  /* Segment freigeben        */
    char * addr;                 /* Segmentadresse           */

/* liefert 0 bei Erfolg oder -1 bei Fehler                   */


int shmctl(segid, cmd, sbuf)     /* Segment manipulieren     */
    int segid;
    int cmd;                     /* Kommando                 */
    struct shmid_ds * sbuf;      /* Zeiger auf Status-Puffer */

/* liefert 0 bei Erfolg oder -1 bei Fehler                   */
```

Mit **shmget** übersetzt man einen Segmentschlüssel in eine Segmentnummer. Wenn in **flags** das **IPC_CREAT**-Bit gesetzt ist, wird das Segment gegebenenfalls erzeugt. Die neun am weitesten rechts stehenden Bits von **flags** ergeben den Zugriffsschutz; Leserecht erlaubt nur das Holen der Daten aus dem Segment, während Schreibrecht sowohl das Holen wie auch das Speichern der Daten erlaubt. Da das tatsächliche Lesen und Schreiben außerhalb des Kerns stattfindet, wird der Zugriffsschutz nur wirksam, wenn die Hardware das unterstützt (wie zum Beispiel das Kennzeichnen eines Datensegments als nur lesbar).

Mit **shmat** (*attach*) bildet man ein Segment in seinen eigenen Adreßraum ab. Hierbei kann man mit **addr** auch eine bestimmte Adresse anfordern. Das ist wichtig, wenn man sich gleichzeitig mittels der Systemaufrufe **brk** und **sbrk** (siehe Abschnitt 9.3) dynamischen Speicher holt, da diese ein im Speicher abgebildetes gemeinsames Segment nicht berücksichtigen. Unglücklicherweise bedingt diese Einschränkung,

daß man in vielen Anwendungen über die Benutzung des Speichers Bescheid wissen muß. Außerdem muß man sehr vorsichtig sein bei der Zuweisung von Speicherbereichen an gemeinsame Segmente und an Unterprogramme, die dynamischen Speicher verwalten (wie etwa **malloc**). Dadurch werden Programme fehlerhaft, schwer zu pflegen und nichtportabel. Wenn es für Sie keine Rolle spielt, wohin der Speicher abgebildet wird (Sie Glücklicher!), so können Sie Null als Wert für **addr** benützen und den Kern eine Adresse aussuchen lassen. Allerdings sollten Sie dann nicht erwarten, jemals wieder **brk** oder **sbrk** benutzen zu können. Haben Sie **brk** oder **sbrk** bereits aufgerufen, so wird der Kern ihre Arbeit akzeptieren und das gemeinsame Segment so anlegen, das es keinen Kummer bereitet.

shmat liefert einen ganz biederen Zeiger auf die Anzahl Bytes, die Sie angefordert haben. Sie können dann Daten holen und speichern, indem Sie ganz gewöhnliche C-Operatoren benützen. Unter der Annahme, daß ein anderer Prozeß das gleiche Segment (mit dem gleichen Schlüssel) in seinen Adreßraum abgebildet hat oder abbilden wird, können beide Prozesse dann völlig ungehindert Daten miteinander austauschen.

Hat man den Zugriff auf ein Segment beendet oder benötigt man das verwendete Segmentierungsregister für ein anderes Segment, ruft man **shmdt** (*detach*) auf, um es freizugeben. Der aktuelle Speicherinhalt bleibt davon unberührt; man kann das Segment später wieder abbilden — allerdings kann dann die Adresse in Ihrem Datensegment eine andere sein. Mit einem Aufruf von **shmctl** mit dem Kommando **IPC_RMID** zerstört man schließlich das Segment. Dies ist allerdings ein schwerer Fehler, solange ein Segment im Adreßraum abgebildet ist.

Auf einer Maschine mit begrenztem Adreßraum kann ein Prozeß seinen effektiven Adreßraum vergrößern, indem er Shared-Memory Segmente benutzt. Wenn zum Beispiel der Adreßraum auf 64K (also acht Segmente zu je 8K) limitiert ist, spart man vielleicht ein Segmentierungsregister ein (was einem noch 56K Speicher läßt) und benutzt es, um auf mehrere gemeinsame Segmente zuzugreifen, die mit **shmget** erzeugt wurden. Haben Sie etwa fünf Stück davon, so kann Ihr Programm dann 106K RAM adressieren, indem es mit **shmat** und **shmdt** zwischen den gemeinsamen Segmenten herumwandert. Um so etwas zu machen, muß man schon ziemlich verzweifelt sein, doch unter gewissen Umständen — wie etwa bei einem DBMS mit großem Appetit auf Puffer — kann dies das Leben retten. Aber es kann auch das Leben kosten, also seien Sie vorsichtig.

Der Kern gestattet nur den Zugriff auf Shared-Memory; er hilft nicht, Shared-Memory auch effektiv zu benutzen. Es gibt eine reiche Auswahl an Möglichkeiten, Shared-Memory zu verwenden; die verbreiteste vielleicht ist ein Verfahren zur Übermittlung von Messages. Messages über Shared-Memory zu übermitteln geht schneller als über die Systemaufrufe für Messages, die der Kern bietet. Dies hat hauptsächlich zwei Gründe:

- Während der Übertragung ist der Kern nur mit der Behandlung von Semaphoren beschäftigt — dies sind sehr schnelle Operationen. Der Aufwand für eine Message-Queue wird vermieden.

- Es müssen keine Daten vom Adreßraum eines Prozesses zum Kern und zurück zum Adreßraum des anderen Prozesses kopiert werden. Wenn überhaupt kopiert wird, so findet dies komplett in einem einzigen Adreßraum statt, wo gewöhnliche Maschinenbefehle benützt werden. Auf den meisten Computern ist dies deutlich schneller als eine Kopie mit *Context-Switch*. Aufgrund dieser Effizienz ist es möglich, mittels Shared-Memory wesentlich größere Messages zu schicken als mit den vom Kern zur Verfügung gestellten Messages. Viele Datenbank-Managementsysteme benutzen diese erhöhte Kapazität.

Wir werden die Übermittlung von Messages mittels Shared-Memory und Semaphoren zeigen, indem wir **send** und **receive** neu implementieren. Die grundlegende Idee ist die: Ein Segment des Shared-Memory stellt eine Message-Queue der Länge 1 dar. Für jede solche Queue existieren zwei Semaphore, eines für das Senden (**sndsid**) und eines für den Empfang (**rcvsid**). Um eine Message zu schicken, fordert ein Prozeß das Semaphor für Senden an (**P(sndsid)**), und wenn es verfügbar ist, kopiert er die Message in das gemeinsame Segment. Danach gibt er das Semaphor für Empfang frei (**V(rcvsid)**). Um etwas zu empfangen, eignet sich ein Prozeß das Semaphor für Empfang an, kopiert die Message aus dem gemeinsamen Segment und gibt dann das Semaphor für Senden frei.

Da von diesen Aktivitäten angenommen wird, daß sie schnell sind, müssen wir vermeiden, daß **semget** und **shmget** öfter als nötig aufgerufen werden. Aus diesem Grund unterhalten wir eine lokale Tabelle der aktiven Queues, um den Schlüssel einer Message-Queue in die Nummer des Semaphor-Sets und in die Adresse und Nummer des gemeinsamen Segments zu übersetzen. Dies funktioniert ähnlich wie im vorigen Abschnitt, als wir vermeiden mußten, **msgget** zu oft aufzurufen (in der Funktion **openqueue**). Während der ganzen Unterhaltung halten wir das Segment in unserem Adreßraum abgebildet. Hier ist der Programmtext:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <sys/shm.h>

#define MAXMSG 4096
#define MAXOPEN 20
#define BADADDR (char *)(-1)
```

```
                                /* msg-Information finden                */
static BOOLEAN findinfo(key, sndsidp, rcvsidp, addrp, segidp)
    int key;
    int * sndsidp, * rcvsidp, * segidp;
    char ** addrp;
{
    static struct
    {
        int key;                /* Schluessel der Message-Queue          */
        int sndsid;             /* Semaphornummer zum Senden             */
        int rcvsid;             /* Semaphornummer zum Empfangen          */
        char * addr;            /* Adresse des Shared-Memory Segments    */
        int segid;              /* Shared-Memory Nummer                  */
    } sems[MAXOPEN];
    int i, avail;
    extern int errno;
    char * shmat();

    avail = -1;
    for (i = 0; i < MAXOPEN; i++)
    {
        if (sems[i].key == key)
        {
            * sndsidp = sems[i].sndsid;
            * rcvsidp = sems[i].rcvsid;
            * segidp = sems[i].segid;
            * addrp = sems[i].addr;
            return(TRUE);
        }
        if (sems[i].key == 0 && avail == -1)
            avail = i;
    }
    if (avail == -1)
    {
        errno = 0;
        return(FALSE);
    }
    sems[avail].key = key;
    * sndsidp = sems[avail].sndsid = semtran(key);
    * rcvsidp = sems[avail].rcvsid = semtran(key + 10000);
```

```
    if ((* segidp = sems[avail].segid =
         shmget((key_t)key, MAXMSG, 0666 | IPC_CREAT)) == -1)
        return(FALSE);
    if ((* addrp = sems[avail].addr = shmat(* segidp, (char *)0, 0))
         == BADADDR)
        return(FALSE);
    return(TRUE);
}

BOOLEAN send(dstkey, buf, nbytes)     /* Message schicken  */
    int dstkey;
    char * buf;
    int nbytes;
{
    int sndsid, rcvsid, segid;
    char * addr;

    if (!findinfo(dstkey, &sndsid, &rcvsid, &addr, &segid))
        return(FALSE);
    if (nbytes > MAXMSG)
        nbytes = MAXMSG;
    P(sndsid);
    memcpy(addr, buf, nbytes);
    V(rcvsid);
    return(TRUE);
}

BOOLEAN receive(srckey, buf, nbytes)  /* Message empfangen */
    int srckey;
    char * buf;
    int nbytes;
{
    int sndsid, rcvsid, segid;
    char * addr;

    if (!findinfo(srckey, &sndsid, &rcvsid, &addr, &segid))
        return(FALSE);
    if (nbytes > MAXMSG)
        nbytes = MAXMSG;
```

```
    P(rcvsid);
    memcpy(buf, addr, nbytes);
    V(sndsid);
    return(TRUE);
}

void rmqueue(key)  /* Semaphore und Segment loeschen                    */
    int key;
{
    int sndsid, rcvsid, segid;
    char * addr;

    if (!findinfo(key, &sndsid, &rcvsid, &addr, &segid))
        syserr("findinfo");
                                                  /* Fehler ignorieren */
    (void)semctl(sndsid, 0, IPC_RMID, (char *)0);
    (void)semctl(rcvsid, 0, IPC_RMID, (char *)0);
    (void)shmdt(addr);
    (void)shmctl(segid, IPC_RMID, (struct shmid_ds *)0);
}
```

memcpy ist ein Standard-Unterprogramm in System V, das Daten von dem Bereich, auf den das zweite Argument zeigt, an die Stelle kopiert, auf die das erste Argument zeigt. Das dritte Argument ist die Zahl der zu kopierenden Bytes.

Sollte diese Implementierung nicht schnell genug sein, so kann man einige Maßnahmen ergreifen, obwohl diese nicht unbedingt großen Erfolg haben müssen. Eine Möglichkeit ist, das Durchsuchen der Tabelle in **findfifo** ganz zu vermeiden, indem der Benutzer von **send** und **receive** sich mit einem Index in die Tabelle **sems** befaßt, anstatt mit einem abstrakten Schlüssel. Der Benutzer erhält den Schlüssel durch einen Aufruf ganz zu Beginn, analog zu **semtran** bei Semaphoren.

Für noch höhere Geschwindigkeit können Empfänger oder Sender die Message direkt im gemeinsam benutzten Segment berechnen oder verarbeiten, ohne sie mittels **memcpy** zu verlagern. Dadurch bleiben natürlich die Semaphore für einen längeren Zeitraum gesperrt, so daß ein Engpaß resultiert, falls viele Prozesse die gleiche Message-Queue benützen. Dies wäre der Fall, falls das DBMS eine einzige Anfrage-Queue besitzt, und es Daten direkt im gemeinsam benützten Segment verarbeitet. Kein anderer Prozeß könnte eine Message schicken, bis das DBMS die vorige abgearbeitet hat. Zwar würde sich die Antwortzeit erhöhen, doch die Zeit zur Erledigung einer einzelnen Anfrage (wenn man nur die Extrakosten für die Übergabe von Messages betrachtet) würde sich verringern.

Man könnte beide Zeiten verringern, wenn man das DBMS so konstruiert, daß es sich mit mehreren Shared-Memory-Segmenten befaßt, für jeden Klient eines. Um

dies zuwege zu bringen, müssen weitere Veränderungen vorgenommen werden. Zum einen muß das DBMS ein nichtblockierendes **receive** bereitstellen, da es mehrere Queues bedienen muß — es weiß nicht, welche Queue zuerst eine Message empfangen wird. Außerdem sollte es ein weiteres (allgemeines, nicht binäres) Semaphor geben, das dem DBMS sagt, ob Arbeit anliegt, damit das DBMS kein *busy-wait* ausführt, falls alle Queues leer sind. Falls Sie diesen Ansatz wählen, sollten Sie aufpassen, daß Sie nicht über das Ziel hinausschießen und am Ende ein Monolith herauskommt, in dem das DBMS und alle Klienten integriert sind — das Resultat wäre ein gigantischer Prozeß, der die Hälfte des vorhandenen Hauptspeichers für Puffer für das Shared-Memory benötigt. Man erwartet von Ihnen, daß Sie das Betriebssystem das tun lassen, für das es gemacht wurde, und nicht, daß Sie Ihr eigenes Betriebssystem im Benutzer-Adreßraum schreiben!

Eine andere Möglichkeit ist, auf Semaphore ganz zu verzichten (stets eine erfreuliche Vorstellung). Anfragen an die Datenbank und Antworten, die den Status wiedergeben, werden über Messages abgewickelt, doch die eigentlichen Daten werden über Shared-Memory übergeben. Erhält das DBMS eine Anfrage-Message, so weiß es, daß die zur Verfügung gestellten Daten, falls überhaupt, im gemeinsam benützten Segment zur Verfügung stehen. Empfängt ein Klient eine Antwort, so weiß er, daß die verlangten Daten, falls vorhanden, im Segment liegen. Kein Prozeß wird jemals außer der Reihe auf Shared-Memory zugreifen. Ein derartiges Schema wurde in den Bell Laboratorien verwendet, um eine extrem hochleistungsfähige DBMS-Anwendung zu implementieren.[7] Auf einem einzigen PDP-11/70 Computer verarbeitet diese Anwendung Tausende von Transaktionen pro Stunde von einem Netzwerk aus Hunderten von Terminals. Dies ist so schnell, wie es mit einer PDP-11/70 überhaupt geht.

7.7.3 Shared-Memory in Xenix 3

Kommen wir nun zu den Systemaufrufen für Shared-Memory in Xenix 3. Wie üblich sind sie einfacher als ihre Gegenstücke in System V.

```
#include <sd.h>

char * sdget(path, flags, nbytes, perms)  /* Segment abbilden      */
    char * path;                          /* Pfadname              */
    int flags;                            /* Optionen-Flaggen      */
    long nbytes;                          /* Segmentgroesse        */
    int perms;                            /* Zugriffsschutz-Bits   */

/* liefert Segmentadresse oder -1 bei Fehler                       */
```

[7] Siehe "Automated Repair Service Bureau: Software Tools and Components", von R. F. Bergeron und M.J. Rochkind in *Bell System Technical Journal* 61, no. 6 (July-August 1982), 1177–1195.

```
int sdfree(addr)                    /* Segment freigeben          */
    char * addr;                    /* Segmentadresse             */

/* liefert 0 bei Erfolg oder -1 bei Fehler                        */


int sdenter(addr, flags)            /* Zugriff auf Segment beginnen */
    char * addr;
    int flags;

/* liefert 0 bei Erfolg oder -1 bei Fehler                        */


int sdleave(addr)                   /* Zugriff auf Segment beenden */
    char * addr;

/* liefert 0 bei Erfolg oder -1 bei Fehler                        */


int sdgetv(addr)                    /* Segmentversionsnummer holen */
    char * addr;

/* liefert aktuelle Versionsnummer oder -1 bei Fehler             */


int sdwaitv(addr, vnum)             /* auf Versionsnummer warten  */
    char * addr;
    int vnum;                       /* Versionsnummer             */

/* liefert aktuelle Versionsnummer oder -1 bei Fehler             */
```

Shared-Memory in Xenix 3 gleicht Semaphoren insofern, als eine Datei der Länge Null dazu verwendet wird, das Shared-Memory-Segment zu verkörpern. Diese Datei übernimmt die Rolle des Schlüssels, der in System V benützt wird. Man beginnt mit einem Aufruf von **sdget**, um ein gemeinsames Segment der Größe **nbytes** in den eigenen Adreßraum abzubilden. **flags** wird wie das zweite Argument bei **open** verwendet. Folgende Bits sind definiert:

SD_RDONLY	Nur Lesen des Segments erlauben.
SD_WRITE	Lesen und Schreiben erlauben.

SD_CREAT Anlegen des Segments, falls nötig. **perms** wird für die Zugriffsschutz-Bits verwendet. Wie bei System V erlaubt Schreibrecht sowohl Schreiben als auch Lesen.

SD_UNLOCK Mehreren Prozessen den gleichzeitigen Zugriff auf das Segment gestatten. Das Handbuch von Xenix 3 warnt, daß dies den Zugriff bei Computern ohne passende Segmentierungs-Hardware beträchtlich verlangsamt. (Weshalb, werden wir in Kürze sehen). Wird Shared-Memory zur Übergabe von Messages verwendet, so ist diese Option sowieso unerwünscht.

Die Argumente **size** und **mode** werden nur zusammen mit **SD_CREAT** verwendet.

Im Gegensatz zu System V können Sie nicht einfach loslegen und das Segment benutzen, selbst wenn es in Ihren Adreßraum abgebildet ist. Zuerst müssen Sie **sdenter** aufrufen. Dann sollten Sie **sdleave** so bald als möglich aufrufen, auf jeden Fall aber bevor Sie irgendeinen anderen Systemaufruf ausführen. Tun Sie das nicht, so ist es möglich, daß die Daten im gemeinsam benützten Segment zerstört werden. Es gibt zwei Bits für **flags**: **SD_NOWAIT** veranlaßt **sdenter** dazu, mit einem Fehler zurückzukehren, statt zu blockieren, falls ein anderer Prozeß auf das Segment zugreift (und falls außerdem **SD_UNLOCK** nicht gesetzt ist). **SD_WRITE** erlaubt dem hereinkommenden Prozeß, das Segment zu verändern; dies ist eine strengere Aussage, als nur die Erlaubnis zu besitzen, dies zu tun.

Setzen Sie **SD_UNLOCK** bei **sdget** nicht, so besteht ein vermeintlicher Vorteil darin, daß **sdenter** und **sdleave** als Semaphore agieren, da dann nur ein Prozeß auf einmal auf das Segment zugreifen kann. Unglücklicherweise profitiert man davon überhaupt nicht, da man nicht der Notwendigkeit enthoben wird, zum Senden bzw. Empfangen von Semaphoren **P** bzw. **V** aufzurufen, wie wir es vorhin bei der Implementierung von **send** und **receive** gemacht haben. Sender und Empfänger müssen sich immer noch abwechseln. **sdenter** und **sdleave** garantieren nur, daß nicht zwei von ihnen zur gleichen Zeit auf den Speicher zugreifen — es hält weder den Sender davon ab, seine eigenen Daten zu empfangen, noch verhindert es, daß der Empfänger die gleiche Nachricht zweimal erhält. Semaphore verhindern diese Mißgriffe.

In Wirklichkeit liegt der wahre Vorteil von **sdenter** und **sdleave** woanders: Ein Entwickler kann Shared-Memory in Xenix 3 dadurch verwalten, daß er mittels **sdenter** Daten vom Kern in den Adreßraum desjenigen kopiert, der den Zugriff macht, und sie mittels **sdleave** zurückschreibt. Hat man nicht genügend Hardware-Segmentierungsregister zur Verfügung, bleibt einem keine andere Wahl. Sie könnten ja versuchen, mit Hilfe von Shared-Memory Messages zu simulieren, während der Kern Messages benützt, um Shared-Memory zu simulieren!

Die letzten beiden Systemaufrufe sind sehr merkwürdig. Jedesmal, wenn **sdleave** auf ein Segment ausgeführt wird, wird dessen Versionsnummer erhöht. **sdget** holt die aktuelle Versionsnummer. Der Systemaufruf **sdwaitv** blockiert, bis er einen

bestimmten Wert erhält. Damit können Sie den Sender einer Nachricht auf eine Antwort des Empfängers warten lassen: Der Sender holt sich mit **sdgetv** die Versionsnummer (z.B. N), ruft **sdleave** auf und wartet dann in **sdwaitv** darauf, daß die Versionsnummer zu N+2 wird. Eine gute Idee, die bei mehreren Sendern oder mehreren Empfängern jedoch nicht funktioniert.

Wir zeigen Ihnen nun eine neue Implementierung von **send** und **receive**, zum vierten und letzten Mal (jedenfalls in diesem Buch):

```
#include <sd.h>

#define MAXMSG 4096
#define MAXOPEN 20
#define BADADDR (char *)(-1)
                                /* msg-Information finden                */
static BOOLEAN findinfo(key, sndsidp, rcvsidp, addrp)
    int key;
    int * sndsidp, * rcvsidp;
    char ** addrp;
{
    static struct
    {
        int key;                /* Schluessel der Message-Queue          */
        int sndsid;             /* Semaphor-Nummer zum Senden            */
        int rcvsid;             /* Semaphor-Nummer zum Empfangen         */
        char * addr;            /* Adresse des Shared-Memory-Segments    */
    } sems[MAXOPEN];
    int i, avail;
    char sname[20];
    extern int errno;
    char * sdget();

    avail = -1;
    for (i = 0; i < MAXOPEN; i++)
    {
        if (sems[i].key == key)
        {
            * sndsidp = sems[i].sndsid;
            * rcvsidp = sems[i].rcvsid;
            * addrp = sems[i].addr;
            return(TRUE);
        }
```

```
        if (sems[i].key == 0 && avail == -1)
            avail = i;
    }
    if (avail == -1)
    {
        errno = 0;
        return(FALSE);
    }
    sems[avail].key = key;
    * sndsidp = sems[avail].sndsid = semtran(key);
    * rcvsidp = sems[avail].rcvsid = semtran(key + 10000);
    sprintf(sname, "/tmp/sd%d", key);
    if ((* addrp = sems[avail].addr =
         sdget(sname, SD_CREAT, MAXMSG, 0666)) == BADADDR)
        return(FALSE);
    return(TRUE);
}

BOOLEAN send(dstkey, buf, nbytes)  /* Message schicken */
    int dstkey;
    char * buf;
    int nbytes;
{
    int sndsid, rcvsid;
    char * addr;

    if (!findinfo(dstkey, &sndsid, &rcvsid, &addr))
        return(FALSE);
    if (nbytes > MAXMSG)
        nbytes = MAXMSG;
    P(sndsid);
    if (sdenter(addr, SD_WRITE) == -1)
    {
        V(sndsid);
        return(FALSE);
    }
    memcpy(addr, buf, nbytes);
    if (sdleave(addr) == -1)
    {
        V(sndsid);
        return(FALSE);
    }
```

```
    V(rcvsid);
    return(TRUE);
}

BOOLEAN receive(srckey, buf, nbytes)  /* Message empfangen */
    int srckey;
    char * buf;
    int nbytes;
{
    int sndsid, rcvsid;
    char * addr;

    if (!findinfo(srckey, &sndsid, &rcvsid, &addr))
        return(FALSE);
    if (nbytes > MAXMSG)
        nbytes = MAXMSG;
    P(rcvsid);
    if (sdenter(addr, 0) == -1)
    {
        V(rcvsid);
        return(FALSE);
    }
    memcpy(buf, addr, nbytes);
    if (sdleave(addr) == -1)
    {
        V(rcvsid);
        return(FALSE);
    }
    V(sndsid);
    return(TRUE);
}

void rmqueue(key)                    /* Segment freigeben */
    int key;
{
    int sndsid, rcvsid;
    char * addr;

    if (!findinfo(key, &sndsid, &rcvsid, &addr))
        syserr("findinfo");
    (void)sdfree(addr);              /* Fehler ignorieren */
}
```

memcpy ist in Xenix 3 nicht verfügbar, kann jedoch leicht in C geschrieben werden. In **send** und **receive** löschen wir **sndsid** und **rcvsid** falls ein Fehler auftritt. Nur weil wir es nicht geschafft haben, unser Vorhaben durchzuführen, bedeutet das noch nicht, daß wir nicht jemand anderem erlauben, es zu versuchen. Falls gewünscht, können entsprechende Aufrufe von **unlink** dazugefügt werden.

7.8 Record-Locking in Xenix 3

Record-Locking ist die Spitze eines ziemlich großen Eisbergs. Nachdem wir den verborgenen Teil inspiziert haben, zeigen wir Ihnen, wie die Spitze in Xenix 3 implementiert ist.

In einer Datenbank-Anwendung nennt man eine *Transaktion* eine kleine Berechnung, die eine Datenbank auf konsistente Weise verändert. Mit *konsistent* meinen wir, daß die Datenbank vor und nach — jedoch nicht notwendigerweise während — der Ausführung der Transaktion gewissen strukturellen Regeln genügt. Zum Beispiel kann eine Regel besagen, daß ein Angestellter nur einer Abteilung angehören darf. Eine Transaktion für den Umzug eines Angestellten in eine andere Abteilung muß zwei verschiedene Operationen ausführen: das Löschen aus einem Datensatz einer Abteilung und das Einfügen in den Datensatz der neuen Abteilung. Während der Ausführung kann der Angestellte in keiner oder in zwei Abteilungen vorhanden sein; vor und nach der Transaktion muß der Angestellte in genau einer Abteilung enthalten sein.

Um sicherzustellen, daß die Transaktion die Datenbank in einem konsistenten Zustand beläßt, muß diese *atomar* (unteilbar) sein. Das bedeutet, daß sie entweder als Ganzes oder überhaupt nicht ausgeführt werden muß. Weiterhin muß sie *serialisierbar* sein, was bedeutet, daß bei gleichzeitiger Ausführung mehrerer Transaktionen die Wirkung dieselbe ist, wie wenn die Transaktionen einzeln, in beliebiger Reihenfolge, durchgeführt werden.

Unteilbarkeit wird typischerweise dadurch erreicht, daß man die alte Ausführung jedes veränderten Datensatzes in einem *Protokoll zum Zurücksetzen* (dem sogenannten *undo*-Protokoll) aufzeichnet, während die Transaktion durchgeführt wird. Bricht die Transaktion ab, so kann die Datenbank durch Rückgängigmachen jedes veränderten Datensatzes in ihren ursprünglichen Zustand versetzt werden. Ist die Transaktion abgeschlossen, so kann das Protokoll weggeworfen werden.

Serialisierbarkeit erreicht man typischerweise dadurch, daß man die Datensätze sperrt, wenn die Transaktion auf sie zugreift. Es gibt zwei Arten von Sperren: *gemeinsame* und *exklusive*. Eine gemeinsame Sperre wird errichtet, wenn eine Transaktion einen Datensatz liest; sie ist nur dann erlaubt, wenn der Datensatz durch eine andere Transaktion nicht oder nur gemeinsam gesperrt ist. Eine exklusive Sperre wird errichtet, wenn eine Transaktion einen Datensatz schreibt; sie ist nur dann zugelassen, wenn der Datensatz nicht gesperrt ist. Die Sperren dürfen *nicht* abge-

baut werden, wenn der Lese- bzw. Schreibzugriff beendet ist — sie müssen solange aufrecht erhalten werden, bis die Transaktion beendet ist. Solange die Transaktion nicht beendet ist, besteht die Möglichkeit, daß sie rückgängig gemacht wird. Dies kann aber nur dann sicher durchgeführt werden, wenn alle durch die Transaktion berührten Datensätze noch gesperrt sind. Betrachten wir zum Beispiel folgendes Szenario:

1. Transaktion A sperrt den Datensatz 1, speichert eine Kopie im Protokoll zum Zurücksetzen, modifiziert den Datensatz und gibt ihn frei.

2. Transaktion B verändert den Datensatz ebenso (mit anderen Daten).

3. Transaktion B endet.

4. Transaktion A bricht ab und restauriert den Datensatz 1 aus dem Protokoll zum Zurücksetzen. Dadurch werden die durch die Transaktion B vorgenommenen Änderungen aufgehoben.

Man verliert dann keine Änderungen, wenn Transaktion A ihre Sperre auf den Datensatz 1 bis zu ihrem Ende aufrecht erhält. Dadurch wird Transaktion B gezwungen, in Schritt 2 zu warten. Ebenso schlecht wäre es, wenn man kein Protokoll zum Zurücksetzen zur Verfügung hätte. In diesem Fall würden die Auswirkungen einer zur Hälfte ausgeführten Transaktion (die die Datenbank in einen inkonsistenten Zustand setzen kann) nach Abbruch der Transaktion bestehen bleiben. Es gäbe keinen Weg, den Schaden wiedergutzumachen.

Deshalb macht es wenig Sinn, Record-Locking ohne Protokoll zum Zurücksetzen zu verwenden, solange Sie nicht sicherstellen können, daß alle Transaktionen normal enden — dies ist unmöglich. Eine abgebrochene Transaktion, die nur ihre Sperren entfernt, ohne ihre Änderungen rückgängig zu machen, kann ebensoviel Inkonsistenz hervorrufen wie eine Transaktion, die eine Datenbank ganz ohne Sperren verändert. Record-Locking ist kein Lösungsschema zur Kontrolle von gleichzeitigen Zugriffen, sondern nur ein Teil davon.

Zwei Transaktionen, die versuchen, den gleichen Datensatz zu sperren, können eine Deadlock-Situation verursachen. In diesem Fall muß eine der beiden Transaktionen abgebrochen werden, damit die andere weitermachen kann. Die Verwendung des Protokolls zum Zurücksetzen erlaubt es, die Veränderungen des abgebrochenen Prozesses problemlos zurückzusetzen. Es ist ein wenig zeitraubend, herauszufinden, ob eine Deadlock-Situation vorliegt: Man muß die Tabelle der Sperren durchsuchen, um einen Abhängigkeitsgraph zu konstruieren, in dem Zyklen gefunden werden müssen. Auch der Vorgang der *Suche nach einem Opfer* — die Entscheidung, welche der in der Deadlock-Situation beteiligten Transaktionen abgebrochen werden soll — kann aufwendig sein. Zum Beispiel könnte es am besten sein, diejenige Transaktion abzubrechen, die bisher am wenigsten Daten verarbeitet hat, die Transaktion

mit der niedrigsten Priorität oder die Transaktion, die die meisten Datensätze an sich bindet. Die Leistungsfähigkeit des DBMS kann entscheidend davon abhängen, wie oft geprüft wird, ob eine Deadlock-Situation vorliegt, wie oft Suche nach einem Opfer durchgeführt wird, wie gut diese Auswahl ist und wie schnell all diese Operationen durchgeführt werden.

Einige UNIX-Versionen besitzen Record-Locking als Systemaufruf im Kern eingebaut. Die von Xenix 3 und PC/IX verwendeten und von */usr/group* vorgeschlagenen Methoden (alle sind einander ziemlich ähnlich) besitzen folgende Mängel:

- Es gibt kein Protokoll zum Zurücksetzen. Wird eine Deadlock-Situation entdeckt, so muß der Prozeß, der den Fehler empfängt, alles selbst rückgängig machen. Da dies eine komplexe Operation ist, wählen Entwickler normalerweise einen zentralisierten DBMS-Entwurf. Wie wir in Abschnitt 7.2 hervorgehoben haben, besteht bei einem zentralisierten Ansatz keine Notwendigkeit, daß der Kern Record-Locking unterstützt, da dies ebenso wirksam im Benutzer-Adreßraum durchgeführt werden kann.
- Bei *jeder* Anforderung einer Sperre und bei *jedem* Lese- und Schreibzugriff auf eine Datenbank-Datei wird geprüft, ob eine Deadlock-Situation vorliegt.
- Die Suche nach Opfern geht sehr primitiv vonstatten. Der erste Prozeß, der eine Deadlock-Situation herbeiführen würde, falls ihm Zugriff gewährt werden würde, erhält ein Fehlerresultat.
- In Xenix 3 sind nur 200 Sperren möglich.
- In manchen Entwürfen, wie z.B. in PC/IX und im Standard, wie er von */usr/group* vorgeschlagen ist, gibt es keine gemeinsamen, sondern nur exklusive Sperren. In vielen Anwendungen wird dadurch die Zahl gleichzeitiger Zugriffe wesentlich verringert.

Nachdem wir Sie nun gründlich entmutigt haben, durch den Kern unterstütztes Record-Locking zu benützen, zeigen wir Ihnen, wie Record-Locking in Xenix 3 funktioniert. Die Systemaufrufe von PC/IX und von */usr/group* arbeiten ähnlich (bei beiden heißt der Systemaufruf **lockf**).

```
#include <sys/locking.h>

int locking(fd, flags, nbytes)  /* Datensatz sperren oder freigeben */
    int fd;                     /* Filedeskriptor                   */
    int flags;                  /* Optionen-Flaggen                 */
    long nbytes;                /* Zahl zu sperrender Bytes         */

/* liefert 0 bei Erfolg oder -1 bei Fehler                          */
```

Da eine Datei in UNIX nur aus einer Folge von Bytes besteht (ohne das Konzept von Datensätzen), arbeiten diese Systemaufrufe für einen beliebigen Bereich einer Datei, der durch die **nbytes** Bytes definiert ist, die beim aktuellen Positionszeiger beginnen. Um einen Bereich zu sperren oder freizugeben, muß die Datei zuerst geöffnet, und der Positionszeiger mittels **lseek** gesetzt werden. Gerätedateien, Pipes, FIFO-Dateien und Kataloge können nicht gesperrt werden. Die Größe des zu sperrenden oder freizugebenden Bereichs wird durch **nbytes** angegeben. Hat **nbytes** den Wert Null, so wird die gesamte Datei gesperrt bzw. freigegeben. Der Bereich braucht noch nicht existieren, d.h., **nbytes** darf einen Bereich definieren, der über das aktuelle Dateiende hinausgeht. Das Argument **flags** kann einen der folgenden Werte besitzen:

LK_UNLCK	Freigeben des Bereichs.
LK_LOCK	Eine exklusive Sperre auf den Bereich setzen. Falls der Bereich momentan gesperrt ist (gemeinsam oder exklusiv), darauf warten, daß er frei wird.
LK_NBLCK	Ähnlich **LK_LOCK**, doch erfolgt eine Rückkehr mit dem Fehler-Code **EACCESS**, falls der Bereich gesperrt ist.
LK_RLCK	Eine gemeinsame Sperre auf den Bereich setzen. Falls der Bereich bereits exklusiv gesperrt ist, darauf warten, daß er frei wird.
LK_NBRLCK	Ähnlich **LK_RLCK**, doch erfolgt eine Rückkehr mit dem Fehler-Code **EACCESS**, falls der Bereich exklusiv gesperrt ist.

Bei jedem Aufruf, der eine Sperre setzt, wird geprüft, ob eine Deadlock-Situation vorliegt. Würde das Bewilligen der Sperre eine Deadlock-Situation verursachen, so wird der Fehler-Code **EDEADLOCK** zurückgeliefert. Beachten Sie, daß dieses Schema auf der Annahme beruht, daß ein Prozeß, der auf eine Sperre wartet, darauf wartet, daß sie frei wird. Da **locking** durch ein Signal unterbrochen werden kann, kann eine offensichtliche Deadlock-Situation von den darin beteiligten Prozessen beseitigt werden, selbst wenn die Sperre, die die Deadlock-Situation ausgelöst hat, bewilligt wurde. Dies ist jedoch kein Fehler in der Implementierung in Xenix 3, da auch ein zentralisiertes DBMS mit eigenem Lock-Manager unzweifelhaft dieselbe Annahme macht — welche andere Wahl bleibt?

Bereiche, die gesperrt bzw. frei sind, dürfen sich überschneiden. Im Handbuch von Xenix 3 ist ausführlich beschrieben, wie Überschneidungen behandelt werden. In der Realität stellen Bereiche unterschiedliche Datensätze dar, und es treten keine Überschneidungen auf. Überhaupt ist das Behandeln von Überschneidungen Verschwendung, da es die Verarbeitung verlangsamt. Weiterhin ist es zu den meisten Ausführungen von Protokollen zum Zurücksetzen inkompatibel und wird von Datenbank-Managementsystemen nicht verwendet.

Hier ist ein primitives Beispiel, das zeigt, wie **locking** aufgerufen wird. (Sie sollten es nicht als Empfehlung auffassen.) Wir öffnen eine Datei, sperren einen Datensatz, lesen ihn, verändern ihn, schreiben ihn zurück und geben ihn frei.

```
#define POS 200L
#define RECSIZE 100

void locktest()    /* Systemaufruf "locking" testen                    */
{
    int fd, nread;
    char buf[RECSIZE];
    long lseek();

    if ((fd = open("dbfile", O_RDWR, 0)) == -1)
        syserr("open");
    if (lseek(fd, POS, 0) == -1)
        syserr("lseek");
    if (locking(fd, LK_LOCK, RECSIZE) == -1)
        syserr("locking");
    switch (read(fd, buf, RECSIZE))
    {
        case -1:
            syserr("read");
        case RECSIZE:
            break;
        default:
            fatal("read");
    }
    process(buf);  /* anwendungsspezifische Verarbeitung durchfuehren */
    if (lseek(fd, POS, 0) == -1)
        syserr("lseek");
    if (write(fd, buf, RECSIZE) == -1)
        syserr("write");
    if (locking(fd, LK_UNLCK, RECSIZE) == -1)
        syserr("locking");
}
```

7.9 Portabilität

Alles, was in diesem Kapitel besprochen wurde, funktioniert nicht auf UNIX-Systemen, die von Version 7 oder 4.2 BSD abgeleitet sind. **send**, **receive**, **P** und **V** könnten jedoch in 4.2 BSD mit Systemaufrufen implementiert werden, die es nur

bei dieser UNIX-Version gibt. Auf reinen Version 7 Systemen kann jedoch kein leistungsfähiges und zuverlässiges DBMS implementiert werden.[8]

Bei Systemen, die von System III abgeleitet sind, sollte man so weit wie möglich höherentwickelte Funktionen wie **send**, **receive**, **P** und **V** verwenden und nicht direkte Systemaufrufe benutzen. Falls nötig, können die von uns vorgestellten Funktionen in geringem Maße verändert werden. Zum Beispiel könnte man leicht eine Option hinzufügen, die das Blockieren verhindert.

Für den Anfang sollten Sie FIFO-Dateien sowohl für Messages als auch für Semaphore verwenden. Läuft Ihre Anwendung, so sollten Sie den Durchsatz messen, um zu prüfen, ob eine andere Implementierung nötig ist. Falls dem so ist, so modifizieren Sie die Implementierung Ihrer höherentwickelten Funktionen — lassen Sie jedoch die Schnittstelle unverändert. Dadurch können Sie Ihr System auf weitere Umgebungen, die nicht von System III abgeleitet sind, portieren, so daß Ihr System zwar an Effizienz, nicht jedoch an Leistungsumfang verliert.

Viele kommerzielle Entwickler befassen sich nur mit Portierungen zwischen System V und Xenix 3. Für beide Systeme haben wir in diesem Kapitel Hochleistungs-Implementierungen für Messages und Semaphore gezeigt. Wird Shared-Memory benötigt, so kann man auch dafür höherentwickelte Funktionen entwerfen, da sowohl System V als auch Xenix 3 Shared-Memory explizit unterstützen. Wird Shared-Memory jedoch nur für Messages benötigt, so sollte es in der Implementierung von **send** und **receive** untergebracht werden.

Zur Implementierung eines DBMS oder verwandten Anwendungen in UNIX gehört natürlich wesentlich mehr als nur Prozeßkommunikation. Wir haben diesem Thema deshalb so viel Aufmerksamkeit gewidmet, weil es in einem Bereich liegt, für den UNIX am wenigsten adäquat ist. In Bereichen wie Dateizugriff oder Gerätezugriff ist UNIX schon immer offener gewesen. Wenn sich zwar das UNIX-Dateisystem nicht für ein DBMS eignen mag, so können Sie es doch zumindest durch Verwendung nackter Ein- und Ausgabe umgehen (siehe Abschnitt 3.3). Weiterhin spielt es keine Rolle, mit welchen Geräten Sie sich befassen müssen — Sie können immer einen Gerätetreiber dazufügen (vorausgesetzt, der Kern hat Platz für weitere Treiber). Man kann heute sagen, daß moderne UNIX-Systeme, wie System V oder Xenix 3, eine ausgezeichnete Grundlage für DBMS-Implementierungen bieten. Dies bedeutet nicht, daß diese Aufgabe einfach ist, sondern nur, daß sie zu bewältigen ist, und daß die Ergebnisse hervorragend sein können.

[8] Lesen Sie zum Beweis den folgenden Artikel von Michael Stonebraker: "Retrospection on a Database System", *ACM Transactions on Database Systems* 5, no. 2 (June 1980), 225–240. Stonebraker versuchte, ein Mehrprozeß-DBMS auf einem UNIX-System ohne FIFO-Dateien oder Messages zu implementieren.

Aufgaben

7.1 Modifizieren Sie die Shell aus Abschnitt 6.4 so, daß sie FIFO-Dateien statt Pipes verwendet.

7.2 Falls Sie Zugriff auf eine oder mehrere UNIX-Versionen besitzen, die Messages oder Shared-Memory unterstützen (wie System V, Xenix 3 oder 4.2 BSD), so führen Sie ein Experiment durch. Das Experiment soll die Leistungsfähigkeit von Messages vergleichen, die mit unterschiedlichen Mechanismen realisiert sind (FIFO-Dateien, durch den Kern unterstützte Messages und Shared-Memory). Beeinflußt die Länge einer Message das Ergebnis des Vergleichs? Erstellen Sie eine Tabelle, die Ihre Resultate zusammenfaßt.

7.3 Modifizieren Sie die Funktion **receive** so, daß sie ein viertes Argument in Form einer Boole'schen Variablen empfängt. Dieses Argument soll der Funktion mitteilen, ob sie bei einer leeren Queue blockieren soll. Ändern Sie entsprechend alle Implementierungen (FIFO-Dateien, Messages und beide Shared-Memory-Versionen).

7.4 Entwerfen Sie die Message-Systemaufrufe von System V neu, so daß sie einfacher zu verwenden sind, aber die meisten der wichtigen Merkmale erhalten bleiben. Versuchen Sie, Ihren Entwurf mit Hilfe der existierenden Systemaufrufe zu implementieren. Die Komplexität Ihrer neuen Systemaufrufe sollte zwischen **send/receive** und den existierenden Systemaufrufen liegen.

7.5 Wie Aufgabe 7.4, jedoch für Semaphore.

7.6 Wie Aufgabe 7.4, jedoch für Shared-Memory.

7.7 Entwerfen und implementieren Sie einen Lock-Manager für das zentralisierte DBMS aus Abschnitt 7.4.

Kapitel 8: Signale

8.1 Einführung

Signale und Messages sind ziemlich ähnliche Konzepte, jedoch nicht so sehr, daß Signale es verdienen würden, in einem der Kapitel über Prozeßkommunikation vorzukommen. Tatsächlich sind Signale unhandliche Mechanismen zur Prozeßkommunikation, wie wir in diesem Kapitel noch sehen werden. Signale unterscheiden sich von Messages in folgenden Punkten:

- Ein Signal darf zu einem beliebigen Zeitpunkt geschickt werden; gelegentlich stammt es von einem anderen Prozeß, häufiger jedoch vom Kern nach einem außergewöhnlichen Ereignis (z.B. das Abhängen eines Terminals oder ein Fehler bei einer Gleitkommaoperation).

- Man muß ein Signal nicht unbedingt empfangen und darauf reagieren. Per Voreinstellung terminieren die meisten Signale den Zielprozeß. Alternativ kann der Prozeß auch Signale eines bestimmten Typs ignorieren. Selbst wenn ein Prozeß versuchen sollte, ein Signal zu empfangen, so kann dieses dennoch vorbeirutschen und den Prozeß beenden.

- Signale beinhalten keinerlei Information; insbesondere kann der Empfänger nicht die Identität des Senders ermitteln.

- Signale können nur an Prozesse geschickt werden, nicht an eine Message-Queue. Deshalb kann ein Signal nicht als Semaphor verwendet werden.

Warum soll man sich bei diesen Einschränkungen überhaupt mit Signalen befassen? Nun, es ist in erster Linie eine Frage der Verteidigung. Der Kern schickt Signale — und ob man sie nun mag oder nicht, man muß Vorsorge treffen, daß sie ignoriert oder abgefangen werden, da der Prozeß anderenfalls terminiert. Signale sollten jedoch nur bei außergewöhnlichen Ereignissen verwendet werden, niemals zur normalen

Kommunikation. Dafür sollten die in Kapitel 6 und 7 beschriebenen Hilfsmittel eingesetzt werden.

Im nächsten Abschnitt beginnen wir mit einer Liste der verschiedenen Signale, die geschickt werden können, zusammen mit der Erklärung, weshalb sie geschickt werden und welche Wirkung sie auf einen Prozeß haben. Danach erörtern wir den Systemaufruf **signal**, der zur Beschreibung der Aktion verwendet wird, die beim Empfang eines Signals stattfindet. Außerdem beschäftigen wir uns mit den verwandten Systemaufrufen **kill**, **pause** und **alarm**. Wir schließen mit unserem gewohnten Abschnitt über Portabilität, in dem es diesmal nicht viel zu sagen gibt, da fast alles in diesem Kapitel allen modernen UNIX-Versionen, ausgenommen Berkeley 4.2 BSD, gemeinsam ist.

8.2 Signaltypen

In System III und System V sind 19 Signaltypen definiert, von denen die ersten 15 auch in Version 7 enthalten sind. Obwohl diese in mehrere verwandte Gruppen aufgegliedert sind, stellen sie in Wirklichkeit nur eine gemischte Zusammenstellung von Ereignissen dar, für die der Signalmechanismus nützlich ist. Manche Signale werden sehr häufig benutzt, während andere extrem unbekannt sind und nur von ein oder zwei Programmen benutzt werden. Das *Trace-Trap*-Signal z.B. wird nur von den Debuggern **adb** und **sdb** benützt.

Die folgende Zusammenstellung gibt eine kurze Erläuterung jedes Signals. Mit Ausnahme der beiden letzten aufgeführten Signale, die ignoriert werden, ist die Standardreaktion eines Prozesses bei Empfang eines Signals Prozeßabbruch. Hinter den symbolischen Namen, die in der Definitionsdatei **/usr/include/signal.h** vereinbart sind, ist jeweils in Klammern die Signalnummer angegeben. Sie sollten stets den symbolischen Namen und nicht die Signalnummer benützen.

SIGHUP (1) *Leitungsunterbrechung* (*Hangup*). Schaltet man ein Terminal ab oder unterbricht man eine Modem-Verbindung, so wird dieses Signal an jeden Prozeß geschickt, für den dieses Terminal das Kontrollterminal darstellt (siehe Abschnitt 1.5). Das Signal wird außerdem an jeden Prozeß einer Prozeßgruppe geschickt, falls der Prozeßgruppenleiter aus irgendeinem Grund terminiert. Dadurch wird das Abhängen von Terminals simuliert, die physikalisch eigentlich gar nicht abgehängt werden können, wie z.B. bei einem Personal-Computer.

SIGINT (2) *Unterbrechung vom Terminal* (*Interrupt*). Dieses Signal wird an jeden mit einem Kontrollterminal verbundenen Prozeß geschickt, wenn dort die *Interrupt*-Taste gedrückt wird. Mit Hilfe des Systemaufrufs **ioctl** kann man diese Wirkung der *Interrupt*-Taste unterdrücken oder die *Interrupt*-Taste ändern (siehe Abschnitt 4.4.6). Beachten Sie, daß das Unterdrücken der *Interrupt*-Taste und das Ignorieren des Signals vollkommen verschiedene Dinge sind, obwohl die Wirkung (oder das Ausbleiben derselben) auf den Prozeß dieselbe ist.

SIGQUIT (3) *Aufhören* (*Quit*). Ähnlich **SIGINT**; das Signal wird jedoch beim Drücken der *Quit*-Taste (normalerweise *Control-*) geschickt. Gewöhnlich sendet man dieses Signal, um einen Speicherauszug zu erhalten.[1]

SIGILL (4) *Illegale Anweisung* (*Illegal Instruction*). Entdeckt die Hardware eine illegale Anweisung, schickt sie dieses Signal. Auf einigen Computern, insbesondere auf der PDP-11, tritt dieser Fehler auf, wenn ein Prozeß eine Gleitkomma-Anweisung auf einem Computer ohne Gleitkomma-Hardware ausführt. Ein Bücherei-Paket fängt dieses Signal ab und interpretiert die betreffende Anweisung durch Software. Manchmal bricht ein Prozeß ab, der Gleitkommarechnung benützt, wenn beim Binden des zugrundeliegenden C-Programms mittels des **cc**-Kommandos versehentlich die Option **-f** vergessen wurde. Da C-Programme im allgemeinen nicht in der Lage sind, ihre Anweisungen zu modifizieren, deutet dieses Signal selten auf einen wirklichen Programmfehler hin.[2]

SIGTRAP (5) *Trace-Trap.* Dieses Signal wird nach jeder Anweisung geschickt, falls ein Prozeß mit eingeschalteter Ablaufverfolgung mittels **ptrace** ausgeführt wird. (Siehe Abschnitt 9.9).

SIGIOT (6) *I/O-Trap-Instruction.*[3] Dieses Signal wird beim Auftreten von Hardware-Fehlern geschickt. Die genaue Wirkung ist maschinenabhängig und bleibt dem Entwickler vorbehalten. In der Realität wird dieses Signal durch das Standard-Unterprogramm **abort** erzeugt, das ein Prozeß aufruft, um so zu enden, daß ein Speicherauszug produziert wird.

---

[1] Auszüge von Halbleiter-Speichern werden immer noch *core*-dump genannt, und dies wird wohl immer so bleiben.

[2] Ist das Textsegment *impure* (keine Segmentseparierung), so können auch Operationen mit fehlerhaften Zeigern dieses Signal bewirken. A.d.Ü.

[3] *IOT* und *EMT* sind mnemotechnische Namen von PDP-11 Maschinenbefehlen.

SIGEMT (7) *Emulator-Trap-Instruction.*[4] Dieses Signal wird geschickt, wenn ein implementierungsabhängiger Hardware-*Interrupt* auftritt — dies kommt allerdings höchst selten vor.

SIGFPE (8) *Fehler bei Gleitkommarechnung* (*Floating-Point-Exception*). Dieses Signal wird geschickt, wenn die Hardware einen Fehler bei der Gleitkommarechnung entdeckt, wie z.B. eine Gleitkommazahl in fehlerhaftem Format, oder bei Division durch 0. Damit wird beinahe immer ein Programmfehler angezeigt.

SIGKILL (9) *Abbruch* (*Kill*). Der allereinzige sichere Weg, einen Prozeß abzubrechen, da dieses Signal immer tödlich wirkt (es kann weder abgefangen noch ignoriert werden). Man sollte es nur in Notfällen verwenden; statt dessen ist **SIGTERM** vorzuziehen (siehe nächste Seite).

SIGBUS (10) *Bus-Fehler* (*Bus-Error*). Dieses Signal wird geschickt, wenn ein implementierungsabhängiger Hardware-*Interrupt* auftritt. Es bedeutet üblicherweise, daß der Prozeß eine ungerade Datenadresse angesprochen hat, die auf eine Wortkante hätte ausgerichtet sein müssen.

SIGSEGV (11) *Verletzung der Segmentierung* (*Segmentation-Violation*). Dieses Signal wird geschickt, wenn ein implementierungsabhängiger Hardware-*Interrupt* auftritt. Normalerweise bedeutet dies, daß der Prozeß Daten außerhalb seines Adreßraums angesprochen hat.

SIGSYS (12) *Fehlerhaftes Argument in einem Systemaufruf.* Wird nicht benützt.[5]

SIGPIPE (13) *Schreiben auf eine nicht zum Lesen geöffnete Pipe.* Dieses Signal wird an einen Prozeß geschickt, der in eine Pipe schreibt, für die kein Lese-Prozeß existiert. Im allgemeinen bedeutet das, daß der Empfänger ein anderer Prozeß war, der unvorhergesehen terminierte. Dieses Signal wird dazu verwendet, alle Prozesse in einer Pipeline zu beenden: Endet ein Prozeß anormal, so erhält der nächste Prozeß zu seiner Rechten *end-of-file*, und alle Prozesse zur Linken ein **SIGPIPE**-Signal. Beachten Sie, daß die Standard-Shell (**sh**) jeden Prozeß in einer Pipeline zum Vater des Prozesses zu seiner Linken macht.

[4] IOT und EMT sind mnemotechnische Namen von PDP-11 Maschinenbefehlen.

[5] Unter Umständen dient dieses Signal zur Interpretation von Programmen für fremde Betriebssysteme. A.d.Ü.

Folglich ist der Sender nicht der Vater des lesenden Prozesses (es ist genau umgekehrt) — er würde sonst auch nicht vom Tod des lesenden Prozesses benachrichtigt werden.

SIGALRM (14) *Wecker* (*Alarm-Clock*). Dieses Signal wird geschickt, wenn der Wecker eines Prozesses abgelaufen ist. Der Wecker wird mit dem Systemaufruf **alarm** gestellt (siehe Abschnitt 8.7).

SIGTERM (15) *Abbruch durch Software* (*Software-Termination*). Das Standard-Signal für Termination. Es ist das von **kill** gesendete Standard-Signal und wird außerdem beim Anhalten des Systems zum Beenden aller aktiven Prozesse verwendet. Ein Programm sollte so geschrieben werden, daß es das Signal auf seiner Standardreaktion beläßt, oder daß es Aufräumarbeiten (wie das Löschen von temporären Dateien) zügig vornimmt und danach **exit** aufruft.

SIGUSR1 (16) *Für den Benutzer vereinbartes Signal 1.* Dieses Signal kann in Anwendungsprogrammen zur Prozeßkommunikation verwendet werden. Dies wird jedoch nicht empfohlen, und folglich wird das Signal nur selten verwendet.

SIGUSR2 (17) *Für den Benutzer vereinbartes Signal 2.* Ähnlich **SIGUSR1**.

SIGCLD (18) *Tod eines Sohn-Prozesses.* Dieses Signal erhält der Vater beim Tod eines Sohn-Prozesses. Das Signal verhält sich anders als die anderen Signale (es wird später in diesem Abschnitt ausführlich besprochen).

SIGPWR (19) *Neustart bei Stromausfall.* Die genaue Bedeutung ist implementierungsabhängig. Eine Möglichkeit besteht darin, daß das Signal gesendet wird, wenn die Stromversorgung im Begriff ist, auszufallen (wenn die Stromspannung (in den USA) zum Beispiel 100 Volt erreicht hat und weiter fällt). Dem Prozeß verbleibt dann nur sehr wenig Zeit zur weiteren Ausführung. Er sollte im Normalfall seine Aufräumarbeiten vornehmen und terminieren (wie bei **SIGTERM**). Will der Prozeß den Stromausfall (der in Wirklichkeit nur ein momentanes Absacken der Stromspannung sein könnte) überleben, so kann er seine Aufräumarbeiten vornehmen und sich für einige Sekunden schlafen legen. Wacht er später wieder auf, so kann er annehmen, daß die Katastrophe nur ein böser Traum war und die Ausführung wiederaufnehmen. Wacht er nicht wieder auf, so sind keine weiteren Handlungen mehr nötig.

Programme, die vor ihrem Ende noch Aufräumarbeiten zu erledigen haben, sollten Vorkehrungen zum Abfangen der Signale **SIGHUP**, **SIGINT** und **SIGTERM** treffen. Solange das Programm noch nicht zuverlässig ist, sollte man sich über **SIGQUIT** eine Möglichkeit offenhalten, das Programm vom Terminal aus (mit Speicherauszug) zu terminieren. Für die anderen Signale werden weitaus seltener Maßnahmen getroffen; im allgemeinen läßt man sie zur Terminierung des Prozesses übrig. Ein wirklich aufpoliertes Programm wird jedoch alles abfangen was nur möglich ist, Aufräumarbeiten vornehmen, möglicherweise den Fehler aufzeichnen und eine nette Fehlermeldung ausgeben. Vom psychologischen Standpunkt aus gesehen ist eine Fehlermeldung wie "Interner Fehler 53: Informieren Sie den Kundendienst" wesentlich zufriedenstellender als die Meldung "Bus error — core dumped" von der Shell.

Bei einigen Signalen wird neben der Standardreaktion Terminierung auch ein Speicherauszug erzeugt. Diese Signale sind **SIGQUIT**, **SIGILL**, **SIGTRAP**, **SIGIOT**, **SIGEMT**, **SIGFPE**, **SIGBUS**, **SIGSEGV** und **SIGSYS**.

8.3 Der Systemaufruf "signal"

```
#include <signal.h>

int (* signal(sig, fcn))()  /* Reaktion auf Signal vereinbaren */
    int sig;                /* Signalnummer                    */
    int (* fcn)();          /* Aktion bei Empfang des Signals  */

/* liefert vorige Aktion oder -1 bei Fehler                    */
```

Diese Vereinbarungen verwirren praktisch jeden, der sie zum ersten Mal sieht. Sie bedeuten jedoch einfach, daß das zweite Argument von **signal** ein Zeiger auf eine Funktion ist, und daß als Resultat ein Zeiger auf eine Funktion geliefert wird.[6] Daß das zweite Argument ein Zeiger auf eine Funktion ist, die eine **int**-Variable liefert, ist ein historisches Überbleibsel; die Funktion liefert nichts und könnte besser wie folgt vereinbart werden:

```
void (* fcn)();
```

(**void** ist ein verhältnismäßig neuer Zusatz zur Sprache C).
Das erste Argument, **sig**, ist eine Signalnummer. Das zweite Argument, **fcn**, kann eines der folgenden drei Dinge sein:

[6] Siehe Abschnitt 8.4 im "*C Reference Manual*" (Anhang A von *The C Programming Language*).

1. **SIG_DFL.** Dadurch wird die Standardreaktion für das Signal gesetzt. **SIGCLD** und **SIGPWR** werden ignoriert, jedes andere Signal beendet den Prozeß, der es empfängt. Wartet ein Vater-Prozeß mittels **wait** auf seinen Sohn, so erhält er über den Exit-Status die Information, daß sein Sohn diesem Schicksal erlegen ist anstatt ein normales Ende durch **exit** (siehe Abschnitt 5.6) gefunden zu haben.

2. **SIG_IGN.** Hierdurch wird das Signal ignoriert; der Prozeß wird dagegen immun. Das Signal **SIGKILL** kann nicht ignoriert werden. Im allgemeinen sollte man nur **SIGHUP**, **SIGINT** und **SIGQUIT** ständig ignorieren. Der Empfang anderer Signale sollte zumindest aufgezeichnet werden, da sie anzeigen, daß etwas Außergewöhnliches passiert ist. **SIG_IGN** besitzt in Verbindung mit **SIGCLD** eine weitere Bedeutung, die weiter unten erklärt wird.

3. *Ein Zeiger auf eine Funktion.* Dadurch werden Vorkehrungen zum Abfangen des Signals getroffen; jedes Signal außer **SIGKILL** kann abgefangen werden. Die Funktion wird aufgerufen, wenn das Signal eintrifft.

Die Reaktion eines Vater-Prozesses auf ein Signal wird von seinen Söhnen geerbt. Die Aktionen **SIG_DFL** und **SIG_IGN** bleiben über **exec** hinweg erhalten, für abgefangene Signale wird jedoch wieder **SIG_DFL** eingestellt. Dies ist wesentlich, da die Funktion, die das Signal abfängt, durch neuen Code überschrieben wird. Selbstverständlich kann das neue Programm seine eigenen Signal-Reaktionen setzen.

Mit Ausnahme von **SIGCLD** werden eintreffende Signale nicht in eine Warteschlange eingereiht. Sie werden entweder ignoriert, beenden den Prozeß oder sie werden abgefangen. Dies ist der wesentliche Grund, weshalb Signale zur Prozeßkommunikation ungeeignet sind — eine Nachricht in Form eines Signals könnte verlorengehen, falls das Signal eintrifft, während der betreffende Signaltyp vorübergehend ignoriert wird. Ein weiteres Problem liegt darin, daß eintreffende Signale ziemlich unhöflich sind: Sie platzen herein, egal was gerade vor sich geht, und dies macht eine korrekte Behandlung schwierig, wie wir in Kürze sehen werden.

Das Resultat von **signal** ist die zuvor eingestellte Reaktion auf das Signal. Dies wird benützt, um das Signal wieder so einzustellen, wie es vorher war. Als Beispiel zeigen wir nun den Code der Funktionen **ignoresig** und **entrysig**, die in der Shell von Abschnitt 6.4 benutzt wurden. Erinnern Sie sich daran, daß durch **ignoresig** alle *Interrupt-* und *Quit*-Signale ignoriert werden, und daß **entrysig** sie wieder so einstellt, wie es beim Start des Programms der Fall war. Genau dies wollen wir, wenn wir ein Kommando aus der Shell ablaufen lassen. Beim ersten Aufruf muß **ignoresig** die alten Signale zur späteren Verwendung durch **entrysig** sichern. Die externen Deklarationen in diesem Beispiel sind faszinierend:

```
#include <signal.h>

static int (* entryint)(), (* entryquit)();
#define BADSIG (int (*)())-1  /* -1 cast: Zeiger auf int-Funktion */

void ignoresig()              /* interrupt und quit ignorieren    */
{
    static BOOLEAN first = TRUE;

    if (first)
    {
        first = FALSE;
        entryint = signal(SIGINT, SIG_IGN);
        entryquit = signal(SIGQUIT, SIG_IGN);
        if (entryint == BADSIG || entryquit == BADSIG)
            syserr("signal");
    }
    else if (signal(SIGINT, SIG_IGN) == BADSIG ||
             signal(SIGQUIT, SIG_IGN) == BADSIG)
        syserr("signal");
}

void entrysig()               /* interrupt und quit restaurieren  */
{
    if (signal(SIGINT, entryint) == BADSIG ||
        signal(SIGQUIT, entryquit) == BADSIG)
        syserr("signal");
}
```

Um das Resultat auf einen Fehler zu prüfen, wandeln wir –1 lieber in einen Funktionszeiger um, als daß wir den Resultatwert von **signal** in eine **int**-Variable umwandeln. Der Grund dafür ist, daß einige Versionen des **lint**-Kommandos sich über die zweite, nicht jedoch über die erste Fassung beklagen. Es ist sowieso fragwürdig, bei Funktionen, die Zeiger zurückliefern, –1 als Fehlerresultat zu verwenden (wie auch bei **shmat** in Abschnitt 7.7.2) — wir sind froh, daß es überhaupt funktioniert. Standardfunktionen wie **malloc** benützen stattdessen **NULL**, wie es sich eigentlich auch gehört.

Das Einstellen von Signalen auf die Standardreaktion und das Ignorieren von Signalen ist einfach; schwierig ist das Abfangen. Um Signale abzufangen stellen Sie dem zweiten Argument von **signal** einen Zeiger auf eine Funktion zur Verfügung. Trifft das Signal ein, so passieren zwei Dinge, und zwar in dieser Reihenfolge:

1. Das Signal wird auf die Standardreaktion zurückgesetzt, d.h. üblicherweise Terminierung. Davon ausgenommen sind **SIGILL** und **SIGTRAP**, die nicht zurückgesetzt werden, da sie zu oft hintereinander geschickt werden.

2. Die dafür vorgesehene Funktion wird mit einem einzelnen Integer-Argument aufgerufen, das der Nummer des von ihr abgefangenen Signals entspricht. Wenn die Funktion zurückkehrt (falls überhaupt), so wird die Verarbeitung an dem Punkt wiederaufgenommen, an dem sie unterbrochen wurde.

Falls das Signal eintrifft, während der Prozeß auf irgendein Ereignis wartet, und falls die Funktion, die das Signal abfängt, zurückkommt, so kehrt der unterbrochene Systemaufruf mit dem Fehlerresultat **EINTR** zurück — er wird nicht automatisch neu gestartet. Sie müssen dieses Resultat von einem rechtmäßigen Fehler unterscheiden: Nichts ist wirklich falsch — es ist nur gerade ein Signal eingetroffen. Unter den Systemaufrufen, die imstande sind zu warten, finden sich **open**, **read**, **write**, **wait**, **ioctl**, **pause** und einige der in Kapitel 7 beschriebenen Systemaufrufe zur Prozeßkommunikation (wie **semop** und **msgrcv**). Ein- und Ausgabe für Dateien ist gegen Unterbrechungen geschützt, da sie niemals blockiert.

Wartet ein Vater-Prozeß, der *Interrupt*- und *Quit*-Signale abfängt, auf das Ende eines Sohn-Prozesses, dann besteht die Gefahr, daß der Vater aus **wait** hinausgeworfen wird, bevor der Sohn terminiert, da durch das Drücken einer der beiden Tasten an beide Prozesse ein Signal geschickt wird. Es gibt mehrere Möglichkeiten, dies zu vermeiden, doch die einfachste Methode besteht darin, daß der Vater *Interrupt*- und *Quit*-Signale während der Ausführung von **wait** ignoriert. Dies war bei der Shell in Abschnitt 6.4 der Fall, da der Vater-Prozeß dort die ganze Zeit über *Interrupt*- und *Quit*-Signale ignorierte.

Beim Programmieren ist es extrem schwierig, unterbrochene Systemaufrufe zu berücksichtigen. Entweder muß man so programmieren, daß jeder Systemaufruf, der imstande ist zu warten, neu gestartet wird, oder man ignoriert während der Ausführung eines solchen Systemaufrufs vorübergehend Signale. Beide Methoden sind unpraktisch, und bei der zweiten besteht zusätzlich die Gefahr, ein Signal zu verlieren, während es gerade ignoriert wird. Wir schlagen deshalb folgende Richtlinie vor: *Keine Rückkehr von Funktionen, die Signale abfangen.* Entweder beenden Sie die Ausführung komplett, oder Sie beenden die laufende Operation, indem Sie einen globalen Sprung durchführen (siehe nächsten Abschnitt). Der Texteditor **ed** zum Beispiel fängt *Interrupt*-Signale ab und springt direkt in seine Hauptschleife. Er kümmert sich überhaupt nicht darum, unterbrochene Systemaufrufe wiederaufzunehmen. (Das Abfangen eines Weck-Signals ist eine Ausnahme von dieser Regel; siehe dazu Abschnitt 8.7.)

Gelegentlich verletzt ein UNIX-Programmierer diese Regel dadurch, daß er den schlauen Trick verwendet, beim Eintreffen eines Signals einfach eine Flagge zu setzen, anstatt das Signal richtig abzuarbeiten. Später dann, wenn das Programm dazu bereit ist, prüft er die Flagge, um zu sehen, ob das Signal eingetroffen ist. Dies ist eine schlechte Technik, solange man nicht mit Sicherheit weiß, daß nicht ein wartender Systemaufruf davon unterbrochen wird. Selbst wenn das anfänglich wahr sein sollte, so können nachfolgende Programmänderungen neue Systemaufrufe hinzubringen. Sie können sich die Lage eines anderen Programmierers vorstellen, der versucht herauszufinden, warum Systemaufrufe dauernd schiefgehen, und der nichts von der signalabfangenden Funktion ahnt, die in einem ganz anderen Modul versteckt sein kann.[7]

Trifft ein Signal ein, so stellt das System (für die meisten Signale) wieder die Standardreaktion (Terminierung) ein. Deshalb kann ein unmittelbar nach dem ersten Signal eintreffendes Signal gleichen Typs den Prozeß beenden, bevor dieser eine Chance hat, die Abfangfunktion auch nur zu beginnen. Dies ist selten, aber möglich, vor allem auf überlasteten Systemen. Die Situation ist so, wie wenn Sie ein rohes Ei in Ihrem Wohnzimmer herumwerfen: Obwohl es beinahe immer gefangen wird, ist die Möglichkeit, den Teppich zu beschmutzen, immer noch zu groß. Sie werden sich nur an die *Fehlschläge* erinnern, nicht an die *geglückten Fangversuche.*

Dieses Schlupfloch kann man enger machen, jedoch nicht eliminieren, wenn man sofort nach Eintritt in die Abfangfunktion das Signal ignorieren läßt, bevor man irgend etwas anderes macht. Da wir Signale nicht als Nachrichten verwenden, kümmert es uns in diesem Fall nicht, wenn ein hereinkommendes Signal dabei verlorengeht. Wir sorgen nur dafür, daß das erste Signal korrekt verarbeitet wird und das Programm nicht zu früh terminiert. Hier ist nun eine Funktion, die dafür sorgt, daß *Hangup-*, *Interrupt-*, *Quit-* und Terminierungs-Signale abgefangen werden, so daß eine temporäre Datei gelöscht werden kann, bevor der Prozeß zu Ende geht. Beachten Sie, daß wir uns nur damit befassen, ein Signal abzufangen, das nicht bereits ignoriert wird; anderenfalls würde ein in der Shell im Hintergrund laufender Prozeß (Kommandozeile mit & beendet), bei dem *Interrupt*-Signale ignoriert werden, (nach Erledigung der Aufräumarbeiten) terminieren, wenn wir die *Interrupt*-Taste betätigen.

```
#include <signal.h>

#define BADSIG (int (*)())-1
```

[7] Wenn Sie es sich zur Gewohnheit machen, stets den Wert von **errno** auszugeben, wenn ein Systemaufruf scheitert (zum Beispiel durch Aufruf von **syserr**), würden Sie nicht lange suchen, da der Fehler-Code **EINTR** sofort klärt, was vor sich geht.

```
void catchsigs()                /* Signale abfangen          */
{
    void cleanup();

    setsig(SIGHUP, cleanup);
    setsig(SIGINT, cleanup);
    setsig(SIGQUIT, cleanup);
    setsig(SIGTERM, cleanup);
}

static void setsig(sig, fcn)  /* falls Standardreaktionn   */
    int sig;                    /* Signal setzen             */
    void (* fcn)();
{
    switch (signal(sig, SIG_IGN))
    {
        case BADSIG:
            syserr("signal");
        case SIG_IGN:
            return;
        case SIG_DFL:
            break;
        default:
            fatal("signal already caught!");
    }
    if (signal(sig, fcn) == BADSIG)
        syserr("signal");
}

void cleanup(sig)               /* raeume auf und terminiere */
    int sig;
{
    if (signal(sig, SIG_IGN) == BADSIG)
        syserr("signal");
    if (unlink(TEMPFILE) == -1)
        syserr("unlink");
    switch (sig)
    {
        case SIGHUP:
            fprintf(stderr, "Hangup.\n");
            break;
```

```
            case SIGINT:
                fprintf(stderr, "Interrupt.\n");
                break;
            case SIGQUIT:
                fprintf(stderr, "Quit.\n");
        }
        exit(1);
    }
```

Der erste Aufruf von **signal** in **setsig** dient in Wirklichkeit zwei Zielen:

- Wurde das Signal bereits ignoriert, so wird sichergestellt, daß es auch dabei bleibt.
- Wurde das Signal bisher nicht ignoriert, so wird das Signal ignoriert, so daß wir eine Verschnaufpause haben, in der wir es abfangen lassen können.

catchsig ist eine praktische Funktion zum Aufbewahren. Sie sollten sie am Anfang jedes Programms aufrufen, das vor seinem Ende gewisse Aufräumarbeiten zu erledigen hat. Für die einzelnen Anwendungen können Sie unterschiedliche Definitionen von **cleanup** zur Verfügung stellen.

Das Signal **SIGCLD** verhält sich nicht so wie die anderen Signale. Es gestattet, daß man effizienter auf die Sohn-Prozesse warten kann, falls der Sohn-Prozeß terminiert, bevor der Vater **wait** (siehe Abschnitt 5.6) ausführen kann, oder falls der Vater niemals die Absicht hat, **wait** auszuführen. Normalerweise belegt jeder dieser Sohn-Prozesse immer noch einen Platz in der Prozeßtabelle als Zombie-Prozeß; der Platz wird frei, wenn schließlich jemand auf sie wartet. Wenn jedoch **SIGCLD** beim Vater auf **SIG_IGN** gesetzt wird, so ändern sich die Wirkungen von **exit** und **wait**: Ruft der Sohn **exit** auf, so bleibt kein Zombie-Prozeß übrig — der Platz in der Prozeßtabelle wird sofort freigemacht. Führt der Vater **wait** aus, so wartet **wait** auf *alle* Sohn-Prozesse und liefert anschließend den Fehler-Code **ECHILD**. Die Exit-Codes der Sohn-Prozesse stehen dem Vater dann nicht zur Verfügung. Diese Methode spart beträchtlichen Aufwand, wenn ein Vater viele Söhne ins Leben ruft, die zu verschiedenen Zeitpunkten terminieren können, vorausgesetzt, daß der Vater nicht an ihren Fehler-Codes interessiert ist.

Man kann **SIGCLD** auch abfangen — da es jedoch keinen Weg gibt, den Exit-Code zu bestimmen, ergibt dies wenig Sinn.

Die Handbücher von System III und System V warnen davor, daß **SIGPWR** und **SIGCLD** in zukünftigen UNIX-Versionen geändert werden können, so daß sie in neuen Programmen nicht benutzt werden sollten. Dies klingt nach einem hervorragenden Ratschlag.

8.4 Globale Sprünge

Bei interaktiven Programmen möchte man oft, daß ein Signal wie **SIGINT** nur die gerade laufende Aktion beendet, nicht gleich das ganze Programm. Irgendwie muß die Funktion, die das Signal abfängt, in die Hauptschleife springen, oder wo immer die Ausführung wieder aufgenommen werden soll. Genau das macht **ed**, wenn Sie während einer Editorsitzung die *Interrupt*-Taste drücken.

Wir beginnen wieder einmal damit, falsch anzufangen. Im interaktiven Status-Programm von Abschnitt 3.12 enthält die Funktion **mainloop** die Endlosschleife, die Kommandos verarbeitet. Nehmen Sie an, daß wir **mainloop** nochmals ausführen wollen, wenn der Benutzer die *Interrupt*-Taste drückt. Hier steht, wie man es *nicht* machen sollte:

```
...
void mainloop();

...
if (signal(SIGINT, mainloop) == BADSIG)
    syserr("signal");
mainloop();
...
```

Nach den Vorkehrungen zum Abfangen von *Interrupt*-Signalen rufen wir **mainloop** auf. Trifft ein *Interrupt*-Signal ein, rufen wir **mainloop** erneut auf, dieses Mal als die Funktion, die das Signal abfängt. Eine Weile lang scheint alles gut zu gehen, doch diese Methode beinhaltet zwei schwerwiegende Fehler:

- Jedes Mal, wenn die *Interrupt*-Taste gedrückt wird, bleiben wir in der abfangenden Funktion und verschachteln uns tiefer und tiefer, mit immer weiteren lokalen Variablen und Rücksprungadressen auf dem Stack.
- Kehrt **mainloop** jemals zurück, so fallen wir in die vorhergehende Aufrufstufe zurück — diejenige, die unterbrochen wurde. Das wird den Benutzer — und vermutlich auch den Programmierer — völlig verblüffen.

Die Lösung besteht darin, den Stack in den Zustand zurückzusetzen, den er beim ersten Aufruf von **mainloop** besaß, und die Ausführung durch Aufruf von **mainloop** wiederaufzunehmen. Zur Durchführung genau dieser Aufgaben sind zwei Standard-Unterprogramme (*keine* Systemaufrufe) vorgesehen:

```
#include <setjmp.h>

int setjmp(jmpenv)          /* Ziel fuer longjmp setzen */
    jmp_buf jmpenv;         /* gerettete Umgebung       */

/* liefert 0 oder den durch longjmp gelieferten Wert   */

void longjmp(jmpenv, val)   /* an das durch setjmp      */
                            /* gesetzte Ziel springen   */
    jmp_buf jmpenv;
    int val;                /* Resultatwert fuer setjmp */
```

An der Stelle, an die **longjmp** nach Ihrem Wunsch zurückkehren soll, rufen Sie **setjmp** auf; hierbei wird Null geliefert. Der aktuelle Zustand des Stacks wird in **jmpenv** gespeichert. Wird **longjmp** aufgerufen, so wird der Stack in den Zustand zurückversetzt, in dem er beim Aufruf von **setjmp** war; hierbei werden alle Beweise für irgendwelche Funktionen vernichtet, die nach **setjmp** aufgerufen wurden. Dann kehrt **setjmp** mit dem Wert als Resultat zurück, der **longjmp** als Argument **val** zur Verfügung gestellt wurde (obwohl **setjmp** gar nicht aufgerufen wurde).[8] Wird **setjmp** in Verbindung mit **signal** benützt, so wird **setjmp** im allgemeinen gerade vor der Stelle aufgerufen, an der das Signal auf Abfangen gesetzt wird, und **longjmp** wird aus dem Innern der abfangenden Funktion heraus aufgerufen. Führt **setjmp** dann seinen durch **longjmp** verursachten Rücksprung aus, so wird wiederum **signal** aufgerufen, um das Signal erneut abzufangen (erinnern Sie sich daran, daß ein abgefangenes Signal auf **SIG_DFL** zurückgesetzt wird).

Hier ist der Anfangsteil von **statutil**, der neu geschrieben wurde, um die Signalverarbeitung sauber zu erledigen:

```
#include <signal.h>
#include <setjmp.h>

static jmp_buf jmpbuf;
#define BADSIG (int (*)())-1

main()  /* statutil */
{
    void jumper();
    BOOLEAN catchint;
```

[8] Selbstverständlich darf die Funktion, von der aus **setjmp** aufgerufen wurde, noch nicht beendet sein, wenn **longjmp** aufgerufen wird. A.d.Ü.

```
    catchint = signal(SIGINT, SIG_IGN) != SIG_IGN;
    setbuf(stdout, NULL);
    help();
    if (catchint)
    {
        if (setjmp(jmpbuf) != 0)
            printf(" ... INTERRUPT\n");
        if (signal(SIGINT, jumper) == BADSIG)
            syserr("signal");
    }
    mainloop();
}

static void jumper()  /* im Falle eines Interrupts */
{                     /* zur Hauptschleife         */
    longjmp(jmpbuf, 1);
    fatal("longjmp returned");
}
```

Beachten Sie den Aufruf von **fatal** in der Funktion **jumper**. Eigentlich ist es unmöglich, daß **fatal** aufgerufen wird, doch bei der Fehlersuche passiert oft das Unmögliche. Sie können diesen Teil später immer noch herausnehmen.

Ein guter Programmierer wird **setjmp** und **longjmp** nur sparsam verwenden, da damit eine nichthierarchische Programmstruktur auferlegt wird; ist "**goto**" schlecht, so ist "**longjmp**" noch viel schlechter. Auf der anderen Seite ergibt vielleicht Unrecht und Unrecht zusammen doch Recht: Signale sind von vorneherein nichthierarchisch, so daß **longjmp** die Dinge nur wieder ins Lot bringt.

8.5 Der Systemaufruf "kill"

```
int kill(pid, sig)  /* Signal schicken                    */
    int pid;        /* Nummer des empfangenden Prozesses */
    int sig;        /* Signalnummer                       */

/* liefert 0 bei Erfolg oder -1 bei Fehler                */
```

In den vorigen Abschnitten besprachen wir hauptsächlich Signale, die vom Kern als Folge irgendeines außergewöhnlichen Ereignisses erzeugt wurden. Es ist aber auch einem Prozeß möglich, einem anderen Prozeß ein Signal beliebigen Typs zu schicken. **pid** ist die Prozeßnummer des Prozesses, der das Signal empfangen soll, und **sig** ist die Signalnummer. Die effektive Benutzernummer des sendenden Prozesses muß

mit der des empfangenden Prozesses übereinstimmen, anderenfalls muß der Super-User effektiver Benutzer des sendenden Prozesses sein. Diese Regel gilt für alle nachfolgend aufgeführten Fälle.

Ist **pid** gleich Null, so wird das Signal an alle Prozesse innerhalb der Prozeßgruppe des Senders geschickt (siehe Abschnitt 1.5). Dies wird häufig beim **kill**-Kommando (**kill 0**) verwendet, um alle Hintergrundprozesse abzubrechen, ohne auf ihre Prozeßnummern Bezug nehmen zu müssen. Prozesse in anderen Prozeßgruppen (z.B. ein Datenbank-Managementsystem (DBMS), das Sie zufällig gestartet haben) empfangen das Signal nicht.

Ist **pid** gleich –1, so wird das Signal an alle Prozesse geschickt, deren wirkliche Benutzernummer mit der effektiven Benutzernummer des Senders übereinstimmt. Dies ist eine praktische Methode, alle Prozesse abzubrechen, die man selbst besitzt, egal welcher Prozeßgruppe sie angehören. Führt der Super-User **kill** mit –1 als **pid** aus, so werden mit Ausnahme der Systemprozesse (dazu gehören mindestens der *Swap*-Prozeß (Prozeßnummer 0) und **init** (Prozeßnummer 1)) *alle* Prozesse abgebrochen. Damit schickt man beim Anhalten des Systems **SIGTERM** an alle Prozesse.

Ist **pid** schließlich negativ, aber ungleich –1, so wird das Signal an alle Prozesse geschickt, deren Prozeßgruppennummer mit dem Absolutbetrag von **pid** übereinstimmt. Dies ist eine gute Methode, ein Subsystem wie ein DBMS, ein Kommunikations-Programm oder einen Drucker-Spooler zu beenden, die wahrscheinlich alle einer eigenen Prozeßgruppe angehören.

Im wirklichen Leben wird **kill** in 99 % aller Fälle für folgende Zwecke verwendet:

- Um einen oder mehrere Prozesse zu *beenden*. Üblicherweise verwendet man **SIGTERM**, manchmal aber auch **SIGQUIT** oder **SIGIOT**, um einen Speicherauszug zu erhalten.
- Um die Fehlerbehandlung eines neuen Programms zu *testen*, indem man Signale wie **SIGFPE** (Fehler bei Gleitkommarechnung) simuliert.

Aus den in den vorigen Abschnitten erläuterten Gründen verwendet man **kill** praktisch nie dazu, um einen oder mehrere Prozesse nur über irgendein Ereignis zu informieren (z.B. zur Prozeßkommunikation).

Beachten Sie außerdem, daß der Systemaufruf **kill** meistens über das Kommando **kill** ausgeführt wird. Er wird im allgemeinen nicht in Anwendungsprogramme eingebaut.

8.6 Der Systemaufruf "pause"

```
void pause();  /* auf ein Signal warten */
```

Wir sind bereits einigen Systemaufrufen begegnet, die blockiert sind, während sie auf irgendein Ereignis warten, bevor sie gewisse Tätigkeiten abschließen. Zum Beispiel wartet **read** beim Lesen vom Terminal darauf, daß eine ganze Zeile getippt wird. **pause** dient nur zum Warten: **pause** tut nichts anderes, und wartet auch nicht auf etwas Bestimmtes. Da ein ankommendes Signal jedoch jeden blockierten Systemaufruf unterbricht, können wir ebensogut sagen, daß **pause** auf ein Signal wartet. Wenn die Funktion, die das Signal abfängt, zurückkommt, so kehrt auch **pause** zurück, und **errno** ist auf **EINTR** gesetzt. Da dies jedoch die einzige Möglichkeit ist, daß **pause** jemals zurückkehrt, bringt es nichts, dies zu prüfen.

In den meisten Fällen ist der Wecker das Signal, auf das **pause** wartet. Es lassen sich schwerlich andere Anwendungsbeispiele finden, da in anderen Situationen, in denen Warten erforderlich ist, das Warten bereits in den betreffenden Vorgang selbst miteingebaut ist. Beispiele dafür sind das Warten auf Eingabe (**read**), das Warten auf eine Message (**msgrcv**) und das Warten auf das Ende eines Sohn-Prozesses (**wait**).

8.7 Der Systemaufruf "alarm"

```
unsigned alarm(secs)  /* Wecker stellen          */
    unsigned secs;    /* Sekundenzahl           */

/* liefert verbleibende Restzeit in Sekunden...  */
/* ...vom vorhergehenden Aufruf                  */
```

Jeder Prozeß hat in seinem System-Datensegment eine Weckuhr gespeichert. Wenn der Wecker abgelaufen ist, wird das Signal **SIGALRM** geschickt. Ein Sohn-Prozeß erbt den Wert des Weckers seines Vaters, die eigentliche Weckuhr aber wird nicht gemeinsam benützt. Die Weckuhr bleibt über **exec** hinweg gesetzt.

Der Systemaufruf **alarm** setzt den Wecker auf die durch **secs** gegebene Sekundenzahl. Zurückgeliefert wird die Restzeit des beim vorhergehenden Aufruf gesetzten Werts; dieser Wert ist 0, falls von der Restzeit nichts mehr übrig ist. Die vorhergehende Einstellung wird dazu verwendet, die Uhr so einzustellen, wie sie vor dem Aufruf von **alarm** war. So geht das Unterprogramm **sleep** vor.

Hat **secs** den Wert 0 ist, so wird der Wecker abgestellt. Dies wird erforderlich, falls der Wecker gestellt ist, aber **pause** oder andere wartende Systemaufrufe zurückkehren, bevor der Wecker klingelt. Dies kann passieren, wenn ein anderes Signal ankommt oder das Ereignis eintrifft, auf das man gewartet hat. Wird der

Wecker nicht abgeschaltet, so wird er später klingeln, gerade dann, wenn Sie es am wenigsten erwarten.[9]

Hier ist ein kurzer geschichtlicher Werdegang von **alarm**: Viele Jahre hinweg bestand bei UNIX ein ernstes Problem darin, daß ein Prozeß beim Lesen vom Terminal hängenbleiben konnte — es gab kein blockierungsfreies Lesen. Ein üblicher Trick bestand darin, einen weiteren Prozeß zu erzeugen, der eine Sekunde schlief und dann ein Signal an den lesenden Prozeß schickte, um **read** zu unterbrechen. Zur Abhilfe dieses Problems erfand man den Systemaufruf **alarm**; er stellte einen einfachen Weg dar, nach einer angemessenen Zeitspanne aus einem **read**-Aufruf auszubrechen, ohne dafür andere Prozesse benutzen zu müssen. Gleichzeitig wurde **pause** eingeführt, so daß **sleep** von einem Systemaufruf in ein Unterprogramm degradiert werden konnte. Später, mit System III, wurde **O_NDELAY** hinzugefügt, um eine angenehmere Methode zur Verfügung zu stellen, **read** blockierungsfrei zu machen.

Um Ihnen zu zeigen, wie **alarm** mit **pause** zusammenarbeitet, stellen wir hier eine Fassung von **sleep** vor (die viel einfacher ist als die wirkliche):

```
void sleep2(secs)  /* secs Sekunden schlafen */
    int secs;
{
    void nullfcn();

    if (signal(SIGALRM, nullfcn) == BADSIG)
        syserr("signal");
    alarm(secs);
    pause();
}

static void nullfcn()  /* nichts tun          */
{
}
```

Der Sinn von **nullfcn** besteht nur darin, sicherzustellen, daß **pause** unterbrochen wird und Weck-Signale nicht mit der Standardreaktion beantwortet oder ignoriert werden.

Es gibt noch eine andere verbreitete Verwendung für **alarm**. Normalerweise würden wir das, was wir im Begriff sind vorzustellen, als zu abstrus ansehen, um es in ein so nüchternes Buch wie dieses aufzunehmen. Da dieses Konzept aber auch

[9] Dies verursachte einen der schwerwiegendsten Fehler, den ich jemals hatte. Zu zufälligen Zeitpunkten lieferten Leseoperationen vom Terminal unsinnige Werte. Ich war mit meinem Latein am Ende — und beinahe soweit, einen Diagnosemonitor an die Übertragungsleitung meines Terminals anzuschließen! Schließlich stieß ich dann auf den Wecker, den ich vergessen hatte, abzustellen.

in dem beliebten Texteditor **vi** (aus Berkeley) benutzt wird, werden wir es dennoch beschreiben.

Dieses Konzept löst das folgende Problem: Viele Bildschirmterminals übertragen eine *Escape*-Sequenz, wenn eine Funktionstaste wie 'HOME' oder 'END' gedrückt wird. Solch eine Sequenz besteht aus dem *Escape*-Zeichen (oktal 033), gefolgt von einer Reihe zusätzlicher Zeichen, die angeben, welche Taste gedrückt wurde. Bei einem bestimmten Terminal zum Beispiel werden folgende Sequenzen erzeugt ('ESC' steht für das *Escape*-Zeichen):

Taste	*Escape-Sequenz*
HOME	ESC [H
END	ESC [Y

Es ist leicht, eine Eingabefunktion zu schreiben, die mit Hilfe einer endlichen Maschine Funktionstasten erkennt. Man kann spezielle Symbole definieren, um sie von normalen Zeichen zu unterscheiden. Doch was passiert, wenn die Anwendung ebenfalls das *Escape*-Zeichen verwendet? Wenn der Benutzer *Escape* drückt, woher soll dann die Eingabefunktion wissen, ob sie das Zeichen einfach weiterreichen oder den Rest der *Escape*-Sequenz lesen soll? Der Eingabefluß ist mehrdeutig.

Hier ist die von **vi** gewählte Lösung: Wird ein *Escape*-Zeichen gelesen, so wird die Weckuhr auf eine Sekunde gesetzt und dann ein **read**-Aufruf auf das nächste Zeichen abgesetzt. Kehrt **read** zurück, bevor die Uhr abgelaufen ist, so wird angenommen, daß Zeichen nur deshalb so schnell übertragen werden konnten, weil sie von der Tastatur erzeugt wurden, indem eine Funktionstaste gedrückt wurde; klingelt der Wecker (und unterbricht dabei **read**), so nimmt man an, daß die Zeichen langsam von einem Menschen getippt wurden — daß also wirklich das *Escape*-Zeichen selbst gedrückt wurde.

Das Problem dabei ist, daß die Eingabefunktion ausgetrickst werden kann, wenn ein Mensch schnell genug tippt, weil sie dann glaubt, daß eine Funktionstaste gedrückt wurde. Ist andererseits das System überlastet, so wird die durch eine Funktionstaste erzeugte *Escape*-Sequenz derart langsam übertragen, daß die Eingabefunktion annimmt, daß jedes Zeichen explizit getippt wurde. Das Problem des schnellen Tippers wird verbessert, wenn alle *Escape*-Sequenzen ein seltenes Zeichen gemeinsam haben, wie etwa '[' hinter dem *Escape*-Zeichen. Dann könnte man nämlich die Eingabefunktion so konstruieren, daß alle anderen Kombinationen – egal wie schnell sie getippt werden — als separate Tastendrücke behandelt werden. Nur wenn der Tipper nach dem *Escape*-Zeichen sehr schnell '[' eingibt, ist eine falsche Auslegung möglich. Das Problem beim überlasteten Rechner ist ernsterer

Natur; **vi**-Benutzer sind damit vertraut, wie mangelhaft **vi** auf einem stark belasteten Computer funktioniert (die Erkennung von Funktionstasten ist nur eines der Ärgernisse). Eine perfekte und offensichtliche Lösung für die Anwendung liegt natürlich darin, die *Escape*-Taste einfach nicht zu benutzen. Dann nämlich wird kein Wecker benötigt, und die *Escape*-Sequenzen können durch eine schnelle, unzweideutige endliche Maschine erkannt werden.

Damit Sie sehen, wie **vi** arbeitet, zeigen wir Ihnen hier ein Beispiel einer Eingabefunktion und ein Aufrufprogramm, das wir zu ihrem Test verwendeten. Es werden zwar nur zwei Funktionstasten erkannt ('HOME' und 'END'), doch Erweiterungen zur Behandlung einer ganzen Tastatur sind unkompliziert.

```
#define HOMEKEY -2
#define ENDKEY -3
#define ESC '\33'

main()  /* getkey testen */
{
    int c;

    setraw();
    while (c = getkey())
        switch (c)
        {
            case ESC:
                printf("ESCAPE\r\n");
                continue;
            case HOMEKEY:
                printf("HOME\r\n");
                continue;
            case ENDKEY:
                printf("END\r\n");
                continue;
            case 'q':
                restore();
                exit(0);
            default:
                printf("%c\r\n", c);
        }
}
```

```
int getkey()  /* Zeichen oder Funktionstaste vom Terminal lesen */
{
    static BOOLEAN first = TRUE;
    static char savec = '\0';
    char c;
    enum {NEUTRAL, FINDLB, FINDLTR} state = NEUTRAL;
    extern int errno;

    if (first)
    {
        first = FALSE;
        catchsig();
    }
    if (savec != '\0')
    {
        c = savec;
        savec = '\0';
        return(c);
    }
    while (1)
        switch (read(0, &c, 1))
        {
            case -1:
                if (errno == EINTR)
                    return(ESC);
                syserr("read");
            case 0:
                fatal("Mysterious EOF");
            default:
                switch (state)
                {
                    case NEUTRAL:
                        if (c == ESC)
                        {
                            state = FINDLB;
                            alarm(1);
                            continue;
                        }
                        else
                            return(c);
```

```
                    case FINDLB:
                        alarm(0);
                        if (c == '[')
                        {
                            state = FINDLTR;
                            continue;
                        }
                        else
                        {
                            savec = c;
                            return(ESC);
                        }
                    case FINDLTR:
                        switch (c)
                        {
                            case 'H':
                                return(HOMEKEY);
                            case 'Y':
                                return(ENDKEY);
                        }
                        state = NEUTRAL;
                }
        }
}

static void catchsig()  /* Weckuhr abfangen */
{
    if (signal(SIGALRM, catchsig) == BADSIG)
        syserr("signal");
}
```

setraw und **restore** wurden in den Abschnitten 4.5 und 4.4.8 gezeigt.[10] Wir haben **MIN** und **TIME** auf Null gesetzt, um eine schnellere Reaktion zu erhalten, was in diesem Fall absolut entscheidend ist.

Es ist wichtig, die Dinge so zu regeln, daß ein nicht abgefangenes Weck-Signal niemals eintrifft. Um sicherzustellen, das dies niemals passieren kann, setzen wir **SIGALRM** innerhalb der signalabfangenden Funktion zurück. Da auf der Weckuhr keine Zeit verbleibt, während wir uns in dieser Funktion befinden, ist die kleine Zeitspanne, in der **SIGALRM** auf **SIG_DFL** gesetzt ist, harmlos.

In einer früheren Version dieses Programms wurde das Signal nur dann erneut abgefangen, nachdem **read** unterbrochen wurde:

[10] Im obigen Programm steht LB für Left Bracket und LTR für Letter. A.d.Ü.

```
switch(read(0, &c, 1))
{
    case -1:
        if (errno == EINTR)
        {
            if (signal(SIGALRM, catchsig) == BADSIG)
                syserr("signal");
            return (ESC);
        }
}
```

Das Problem dabei war, daß auf einem überlasteten System der Wecker abgelaufen sein konnte, bevor **read** auch nur aufgerufen worden war — ziemlich überraschend, wenn man bedenkt, wie wenig Anweisungen zwischen dem Aufruf **alarm(1)** und dem Aufruf von **read** liegen. Der vorzeitige Weckruf setzte **SIGALRM** auf **SIG_DFL** zurück, war im übrigen aber klar. Als der Wecker das nächste Mal losging, terminierte der Prozeß. Mir gelang es tatsächlich, den Prozeß abzubrechen, als ich rasch auf die Tasten 'HOME', 'END' und 'ESCAPE' schlug, doch nur dann, wenn ich das System durch einen mit hoher Priorität laufenden Hintergrundprozeß lahmlegte, der die CPU in Beschlag nahm. Das Problem verschwand, indem der Aufruf von **signal** verlagert wurde und damit das Schlupfloch geschlossen wurde. Dies verdeutlicht auf drastische Weise, wie verzwickt Signale sein können.

Sobald das einem *Escape*-Zeichen folgende Zeichen gelesen ist, stellen wir den Wecker ab (**alarm(0)**), doch diesmal liegt keine kritische Situation vor. Läutet der Wecker versehentlich, bevor wir Gelegenheit haben, ihn abzuschalten, so passiert nichts Schlimmes, da zwischen **read** und **alarm** keine Systemaufrufe abgesetzt wurden.

Wenn Sie dieses Beispiel ausprobieren, so werden Sie sehen, daß es ziemlich gut funktioniert. Trotzdem sollten Sie zeitempfindliche Algorithmen auf einem Timesharing-Computer vermeiden. Wir können alle lernen, ohne die *Escape*-Taste zu leben.

8.8 Portabilität

Wenn Sie folgende Richtlinien befolgen, können Sie portable Programme schreiben, die Signale verwenden: Lassen Sie die Finger von den implementierungsabhängigen Signalen **SIGIOT**, **SIGEMT**, **SIGBUS** und **SIGSEGV**. Es ist in Ordnung, wenn Sie sie abfangen, um eine Meldung auszugeben; versuchen Sie jedoch nicht, diesen Signalen irgendeine Bedeutung beizulegen. Vermeiden Sie außerdem **SIGCLD** und **SIGPWR**, da das Handbuch davor warnt, daß sie in zukünftigen UNIX-Versionen eventuell nicht mehr verfügbar sind, und da sie in Version 7 nicht vorhanden sind.

In Version 7 ist **kill** weniger einfallsreich als in späteren Versionen — in Version 7 gibt es keine Prozeßgruppen. Dies sollte jedoch kein Problem darstellen, da **kill** üblicherweise über das **kill**-*Kommando* ausgeführt wird (das zwar nicht portabler, doch zumindest nicht in C-Programmen eingebunden ist).

Bauen Sie in Ihre Programme keine zeitlichen Abhängigkeiten ein. In gewisser Hinsicht ist der **vi**-Editor nicht portabel, da er auf einigen Systemen zufriedenstellend arbeitet, auf anderen dagegen nicht, abhängig von der Rechnerlast und den Eigenschaften des jeweiligen Terminal-Gerätetreibers. Jegliche Verwendung von **alarm** sollte mit Argwohn betrachtet werden. Eine berechtigte Verwendung gibt es in Verbindung mit **read**, um die Terminaleingabe blockierungsfrei zu machen. Dies ist tatsächlich portabler als die Verwendung von **O_NDELAY**, da diese Flagge in Version 7 nicht verfügbar ist. Wenn Sie sich jedoch nur mit Systemen befassen, die auf System III basieren, dann fahren Sie mit **O_NDELAY** besser.

Übertragen Sie Programme von Berkeley 4.2 BSD zu anderen Versionen, so werden Sie große Probleme bekommen, da Signale in 4.2 BSD erheblich geändert wurden. Der Systemaufruf heißt **sigvec**, und er ist (wie üblich) wesentlich einfallsreicher als in den Versionen von AT&T (unerledigte Signale werden in eine Warteschlange eingereiht, eintreffende Signale werden während der Ausführung einer Abfangfunktion gesperrt und unterbrochene Systemaufrufe werden automatisch neu gestartet). Die einzige Lösung besteht darin, den ganzen Programm-Code, der sich mit Signalen befaßt, in einem separaten Modul zusammenzufassen, wie wir es in Kapitel 4 bei der Einstellung von Terminals, und in Kapitel 7 bei der Prozeßkommunikation gemacht haben.

Aufgaben

8.1 Modifizieren Sie die Shell aus Abschnitt 6.4: Lassen Sie **SIGCLD** ignorieren und verwenden Sie die am Ende von Abschnitt 8.3 beschriebene vereinfachte Version von **wait**.

8.2 Wählen Sie einen oder mehrere Computer, mit denen Sie vertraut sind, und geben Sie die Informationen an, die durch **setjmp** in der Umgebung **jmp_buf** gesichert werden müssen.

8.3 Das Unterprogramm **longjmp** räumt nur den Stack auf; es befaßt sich nicht mit Speicherplatz, der durch **malloc** oder **realloc** dynamisch angelegt wurde. Diskutieren Sie die Auswirkungen davon. Ist eine automatische Lösung möglich? Ist sie wünschenswert? Schlagen Sie eine mögliche Entwurfsänderung von (wenigstens) **setjmp**, **longjmp**, **malloc**, **realloc** und **free** vor, die sich des Problems annimmt.

8.4 Kann man die auf Shared-Memory basierende Implementierung von **send** und **receive** aus Abschnitt 7.7.2 so abändern, daß anstelle von **P** und **V** die System-

aufrufe signal, kill und **pause** verwendet werden? Erörtern Sie die Vor- und Nachteile dieser Änderung. Ist die Änderung durchführbar, so implementieren Sie sie.

8.5 Implementieren Sie die Standardfunktion **sleep**. (Sie ist wesentlich komplexer als **sleep2** aus Abschnitt 8.7.)

Kapitel 9: Verschiedene Systemaufrufe

9.1 Einführung

Dieses Kapitel beschreibt Systemaufrufe für Administration, Speicherverwaltung und andere diverse Zwecke, die in keines der vorangegangenen Kapitel so richtig hineinpassen. Alle diese Systemaufrufe haben eines gemeinsam: Sie werden so gut wie nie direkt von einem Anwendungsprogramm aufgerufen. Einige von ihnen, wie **mount** oder **sync**, werden durch Ausführen von Kommandos in der Shell aufgerufen; andere, wie z.B. **sbrk**, werden von Standard-Unterprogrammen wie **malloc** verwendet.

Im folgenden besprechen wir zunächst die für jedermann zugänglichen Systemaufrufe und anschließend diejenigen, die nur dem Super-User zur Verfügung stehen.

9.2 Der Systemaufruf "ulimit"

```
long ulimit(cmd, newlimit)  /* Grenzen fuer einen Benutzer-     */
                            /* prozess setzen bzw. bestimmen    */
    int cmd;                /* Kommando                         */
    long newlimit;          /* neue Grenzen                     */

/* liefert momentane Grenze oder -1 bei Fehler                  */
```

Jeder Prozeß besitzt eine Grenze für die Größe von Dateien und für die Größe des Benutzer-Datensegments. **ulimit** bestimmt bzw. setzt diese Grenzen, je nach Wert von **cmd**.

Falls **cmd** den Wert 1 besitzt, so wird die Grenze für die Dateigröße in Einheiten von 512 Byte-Blöcken zurückgeliefert. Erstmals wird diese Grenze bei Konfiguration des Systems gesetzt; ein typischer Wert ist 2048, wodurch Dateien bis zur Größe eines Megabytes möglich sind.

Hat **cmd** den Wert 2, so wird diese Grenze auf den Wert von **newlimit** gesetzt. Jeder Prozeß kann seine Grenze verringern, doch nur der Super-User darf sie auch vergrößern. Die maximal mögliche Größe ist implementierungsabhängig; theoretisch liegt sie bei ungefähr einer Milliarde Bytes (ein wenig größer als die Kapazität der meisten Platten). Eine Verringerung der Grenze mag im Ausbildungsbereich sinnvoll sein, und eine Vergrößerung könnte bei einem Datenbanksystem nützlich sein. Lesen kann man Dateien beliebiger Größe, selbst wenn sie größer sind als das Limit.

Ist der Wert von **cmd** 3, so wird die Grenze für die Größe des Benutzer-Datensegments in Einheiten von Bytes zurückgeliefert. Der Grenzwert kann nicht geändert werden, doch ist die Größe des Datensegments selbst durch **brk** oder **sbrk** änderbar (siehe nächster Abschnitt).

Das folgende Beispiel zeigt eine Funktion, die beide Grenzen ausgibt:

```
void prtlimits()  /* Grenzen ausgeben */
{
    long ulimit();

    printf("File-size limit = %ld\n", ulimit(1, 0L));
    printf("Data-segment limit = %ld\n", ulimit(3, 0L));
}
```

Die Ausgabe auf meinem System lautete:

```
File-size limit = 2048
Data-segment limit = 65142
```

Zusätzlich zu diesen beiden Beschränkungen besitzt ein Prozeß noch weitere, wie z.B. eine Schranke für die Zahl geöffneter Dateien. Darüberhinaus gibt es auch für die Benutzer gewisse Einschränkungen, wie etwa die Zahl möglicher Prozesse. Schließlich gibt es auch noch Schranken, die systemweit Geltung haben. Jedoch können all diese Schranken nur durch ein erneutes Konfigurieren des Systems geändert werden. Einige Schranken, wie die Zahl möglicher Prozesse oder die Zahl geöffneter Dateien, können nicht über ein Programm bestimmt werden — in diesen Fällen sollte der Systemadministrator befragt werden. Folglich ist die Verwendung solcher Grenzen nicht portabel.[1]

[1] Die Benutzergruppe */usr/group* hat in dem von ihr vorgeschlagenen Standard den Versuch unternommen, diese Situation durch Herausgabe eines Verzeichnisses aller Grenzwerte mit den erforderlichen Minimalwerten zu verbessern. Obwohl der Standard noch zu jung ist, als daß er Anhänger gefunden hätte, sind die Minimalwerte ein guter Anhaltspunkt dafür, wie Version 7, System III und System V normalerweise konfiguriert sind.

9.3 Die Systemaufrufe "brk" und "sbrk"

```
int brk(newbrk)      /* break-Wert setzen        */
    char * newbrk;   /* neuer break-Wert         */

/* liefert 0 bei Erfolg oder -1 bei Fehler      */


char * sbrk(incr)    /* break-Wert aendern       */
    int incr;        /* Inkrement                */

/* liefert alten break-Wert oder -1 bei Fehler */
```

Ein Prozeß startet mit einem Datensegment, das groß genug ist, um die durch den Binder (**ld**) angelegten statischen Variablen aufzunehmen. Die erste Adresse nach dem Datensegment wird *break*-Wert genannt. Der Binder definiert die externe C-Variable **end**, und läßt sie gerade hinter das Datensegment zeigen. Deshalb ist die Adresse von **end** der anfängliche *break*-Wert.

Ein Prozeß kann sein Datensegment vergrößern oder verkleinern, indem er **brk** mit dem gewünschten neuen *break*-Wert als Argument aufruft. Die Schranke für die Größe des Datensegments bestimmt den maximal möglichen *break*-Wert und ist implementierungsabhängig. Diese Schranke reicht von ungefähr 64 KB auf 16-Bit Computern bis in den Megabyte-Bereich auf 32-Bit Computern mit virtuellem Speicher.

Wahrscheinlich ist der Texteditor **ed** das einzige UNIX-Programm, das tatsächlich **brk** aufruft, um den *break*-Wert an eine bestimmte Stelle zu setzen — **ed** benutzt **brk** zum Vergrößern oder Verkleinern eines im Speicher befindlichen Vektors von Zeigern auf Zeilen.

Viel leichter arbeitet man mit **sbrk**.[2] Das Argument von **sbrk** ist gerade ein Inkrement, das zum *break*-Wert addiert werden soll. Als Resultat wird der alte *break*-Wert (also die Anfangsadresse der neuen Speicherfläche) geliefert. **sbrk** wird von **malloc**, dem Standard-Unterprogramm für dynamische Speicherverwaltung, verwendet. Um Speicher zurückzugeben, wird **sbrk** nicht von **free** aufgerufen — **free** hängt die Speicherfläche einfach in eine freie Liste ein. **malloc** ruft **sbrk** für mehr Speicher nur dann auf, wenn es keinen genügend großen Speicherblock mehr in der freien Liste findet.

Da die Position von **end** durch den Binder festgelegt ist, kann sie nach Aufruf von **brk** oder **sbrk** nicht mehr zur Bestimmung des *break*-Werts benutzt werden. Statt dessen kann **sbrk** mit Inkrement 0 aufgerufen werden, um den aktuellen *break* -Wert zu bestimmen, ohne ihn dabei zu ändern.

[2] **brk** und **sbrk** sind in der *System V Interface Definition* nicht enthalten. A.d.Ü.

Um herauszufinden, wieviel Speicherplatz zum Vergrößern des Datensegments verfügbar ist, braucht man nur den aktuellen *break* -Wert von der maximal erlaubten Größe abzuziehen. Hier ist eine praktische Funktion, die die Zahl der verfügbaren Bytes liefert:

```
long avail()  /* Zahl verfuegbarer Bytes ausgeben */
{
    char * sbrk();
    long ulimit();

    return(ulimit(3, 0L) - (long)sbrk(0));
}
```

brk und **sbrk** kehren mit einem Fehler (**ENOMEM**) zurück, falls das Datensegment seine maximal mögliche Ausdehnung erreicht hat. Aus Erfahrung weiß man, daß diese Kontrolle auf einigen UNIX-Implementierungen unzuverlässig ist, insbesondere auf einfachen Computern ohne Hardware-Speicherverwaltung. Ein Programm zur Speicherverwaltung sollte besser **ulimit** zur Bestimmung der Schranke aufrufen, und sich danach selbst die Größe des Datensegments merken, um zu vermeiden, daß **sbrk** jemals mit einer nicht erfüllbaren Forderung aufgerufen wird.

9.4 Der Systemaufruf "umask"

```
int umask(mask)  /* Maske fuer Dateianlage */
    int mask;    /* setzen bzw. bestimmen  */

/* liefert vorige Maske                    */
```

UNIX besitzt eine Sicherheitseinrichtung, die *Maske für Dateianlage* (*file mode creation mask*) genannt wird. Die Maske besteht aus einer neun Bit breiten Zahl, die zum Löschen der entsprechenden Bits in den Zugriffsschutz-Argumenten der Systemaufrufe **creat**, **open** und **mknod** verwendet wird. Glücklicherweise wirkt die Maske nicht auf **chmod**, so daß Prozesse eine Möglichkeit haben, ungeachtet der Maske Zugriffsrechte nach ihren Wünschen zu setzen.

Die Wirkung der Maske für Dateianlage wird am besten durch ein einfaches Beispiel erklärt:

```
void masktest()  /* umask testen */
{
    int fd;

    umask(022);
    if ((fd = creat("temp", 0666)) == -1)
        syserr("creat");
    system("ls -l temp");
}
```

Hier ist die dazugehörige Ausgabe:

```
-rw-r--r--   1 marc staff    0 Oct 12 12:34 temp
```

Wie Sie sehen, sind die Schreib-Bits für die Gruppe und die Welt gelöscht, obwohl **creat** die Aufgabe bekam, jedermann Lesen und Schreiben zu lassen.

Jeder Benutzer bekommt bei der Anmeldung eine voreingestellte Maske für Dateianlage, die bei den meisten Systemen den Wert 22 (oktal) besitzt. Die Shell besitzt ein eingebautes Kommando namens **umask** (wie auch sonst?), das die Maske verändert. Dies wird typischerweise in der Login-Kommandodatei des Benutzers (**.profile**) gemacht, falls die Voreinstellung durch das System nicht passend ist.

Beachten Sie, daß **umask** aus dem gleichen Grund wie **cd** in der Shell eingebaut sein muß. Die Maske für Dateianlage wird von Sohn-Prozessen geerbt, doch die Maske selbst gehört den einzelnen Prozessen. Wäre **umask** ein konventionelles Kommando, so wäre es wirkungslos, da die Shell zu seiner Ausführung einen Sohn-Prozeß erzeugen würde.

Da jeder Prozeß eine Maske besitzt, und da jede Kombination von neun Bits zulässig ist, kann **umask** niemals ein Fehler-Resultat liefern — **umask** liefert als Resultat die alte Maske. Um herauszufinden, welchen Wert die Maske besitzt, ohne sie dabei zu ändern, sind *zwei* **umask**-Aufrufe erforderlich: einer zum Beschaffen des alten Werts (mit einem beliebigen Argument), und ein zweiter, um die Maske auf ihren ursprünglichen Wert zurückzusetzen.

9.5 Der Systemaufruf "ustat"

```
#include <sys/types.h>
#include <ustat.h>

int ustat(dev, ubuf)      /* Information ueber Dateisystem holen */
    int dev;              /* Geraetenummer                       */
    struct ustat * ubuf;  /* Zeiger auf Informations-Puffer      */

/* liefert 0 bei Erfolg oder -1 bei Fehler                       */
```

ustat entnimmt vier Datenfelder aus dem Super-Block eines Dateisystems und legt sie in der zur Verfügung gestellten Struktur **ustat** ab:

```
struct  ustat
{
    daddr_t f_tfree;     /* freie Bloecke (long)         */
    ino_t   f_tinode;    /* freie Inodes (short)         */
    char    f_fname[6];  /* Kennsatz des Dateisystems    */
    char    f_fpack[6];  /* Kennsatz des Plattenstapels */
};
```

In Abschnitt 3.3 haben wir erklärt, wie man den Super-Block liest; wozu brauchen wir dann **ustat**? Platten-Gerätedateien sind normalerweise nur vom Super-User lesbar, während jeder das Recht hat, die Kennsätze und den verbleibenden Speicherplatz eines Dateisystems zu erfahren. Ein Datenbank-Managementsystem z.B. sollte den freien Platz genau überwachen: Es ist sehr unangenehm, eine Folge von Änderungen zu behandeln, die mangels Speicherplatz auf halbem Weg abgebrochen wurde. Auch Subsysteme wie datengewinnende Programme, die riesige Dateien schreiben, müssen den verbleibenden Speicherplatz überwachen, um das Dateisystem nicht zum Überlaufen zu bringen.

Die Kennsätze des Dateisystems und der logischen Platten sind mit keinen anderen Namen verbunden. Beim Montieren des Dateisystems werden sie im Super-Block aufgezeichnet. Ihre Verwendung hängt von der jeweiligen UNIX-Installation ab, so daß sie von keinem portablen Programm benutzt werden sollten.

Hier ist ein Programm, das die durch **ustat** gelieferten Daten ausgibt:

```
#include <sys/types.h>
#include <sys/stat.h>
#include <ustat.h>

prtustat()  /* Informationen ueber Dateisystem ausgeben */
{
    struct stat sbuf;
    struct ustat ubuf;
    char name[7];

    if (stat("/dev/hd0", &sbuf) == -1)
        syserr("stat");
    if (ustat(sbuf.st_rdev, &ubuf) == -1)
        syserr("ustat");
    printf("Free blocks = %ld\n", ubuf.f_tfree);
    printf("Free i-nodes = %ld\n", ubuf.f_tinode);
    name[6] = '\0';
```

```
    strncpy(name, ubuf.f_fname, 6);
    printf("File system label = %s\n", name);
    strncpy(name, ubuf.f_fpack, 6);
    printf("Pack label = %s\n", name);
}
```

Die Ausgabe auf meinem System war wie folgt:

```
Free blocks = 6539
Free i-nodes = 3072
File system label = usr
Pack label = usr001
```

ustat verlangt eine Gerätenummer, die wir durch Aufruf von **stat** (siehe Abschnitt 3.12) erhielten. Beachten Sie, daß die Kennsatzfelder nur genügend Platz zur Unterbringung des sechs Zeichen langen Kennsatzes haben. Wir mußten die Kennsätze in eine lokale Zeichenkette kopieren, um sicherzustellen, daß sie mit einem Nullbyte abgeschlossen sind. Das ist das gleiche Problem, das wir bei den Katalogeinträgen in Abschnitt 3.2 hatten.

9.6 Der Systemaufruf "uname"

```
#include <sys/utsname.h>

int uname(utsbuf)              /* System-Name bestimmen          */
    struct utsname * utsbuf;   /* Zeiger auf Informations-Puffer */

/* liefert nichtnegativen Wert bei Erfolg oder -1 bei Fehler    */
```

uname legt den Namen und die Version des örtlichen UNIX-Systems in dem Puffer ab, auf den **utsbuf** verweist. Die Struktur besitzt folgenden Aufbau:

```
struct utsname
{
    char    sysname[9];   /* System-Name    */
    char    nodename[9];  /* uucp-Node-Name */
    char    release[9];   /* Release-Nummer */
    char    version[9];   /* Versionsnummer */
    char    machine[9];   /* Computermodell */
};
```

Hier ist ein Programm, das die von **uname** gelieferte Information ausgibt:

```
#include <sys/utsname.h>

prtuname()  /* Name und Version des Systems ausgeben */
{
    struct utsname utsbuf;

    if (uname(&utsbuf) == -1)
        syserr("uname");
    printf("System name = %s\n", utsbuf.sysname);
    printf("Node name = %s\n", utsbuf.nodename);
    printf("Release = %s\n", utsbuf.release);
    printf("Version = %s\n", utsbuf.version);
    printf("Machine = %s\n", utsbuf.machine);
}
```

Beachten Sie, daß alle Zeichenketten in der Struktur **utsname** mit einem Nullbyte abgeschlossen sind.

Hier ist die Ausgabe, die ich auf meinem Billigsystem erhielt:

```
System name = PC/IX
Node name = SELF
Release = 1.0
Version = std
Machine = ibmpc-xt
```

Theoretisch könnte man mit Hilfe der Informationen über Release und Version eine Portabilität zwischen System III und System V erreichen, indem man, abhängig von der Version, die Ausführung zur Laufzeit variiert. Eine denkbare Anwendung ist die Implementierung von **send** und **receive** (siehe Kapitel 7) mittels FIFO-Dateien in System III und mittels Messages in System V. Portabilität zu Version 7 (und zurück) läßt sich mit **uname** jedoch nicht erreichen, da dieser Systemaufruf in Version 7 nicht vorhanden ist. Ein weiteres schwerwiegendes Problem ist, daß die von **uname** gelieferte Information nicht standardisiert ist, d.h., es steht den Herstellern frei, ihre eigenen Bezeichnungen für Release und Version zu verwenden. Obwohl PC/IX eine Obermenge von System III ist, wird uns dies in obigem Beispiel weder durch die Release-Nummer noch durch die Version mitgeteilt.

9.7 Der Systemaufruf "sync"

```
void sync()  /* Puffer entleeren, die zur Ausgabe anstehen */
```

Rufen Sie sich aus Abschnitt 2.7 in Erinnerung, daß der Kern die ins Blockdepot geschriebenen Daten nicht sofort auf die Platte schreibt. Daten werden nur dann automatisch auf die Platte geschrieben, wenn der Kern den Puffer für einen anderen Plattenblock benötigt. Der Systemaufruf **sync** entleert alle zur Ausgabe anstehenden Puffer auf die Platte.

Es ist keine gute Idee, Puffer die zur Ausgabe anstehen, zu lange herumliegen zu lassen. Auf den meisten UNIX-Systemen führt der Hintergrund-Prozeß **cron** ungefähr jede Minute einen **sync**-Aufruf aus. So gehen bei einem Zusammenbruch der CPU höchstens Daten der letzten Minute verloren. Will der Systemadministrator das Dateisystem ändern, ohne dabei gestört zu werden, so kann er einfach zuerst **cron** abbrechen. **sync** wird auch beim Anhalten des Systems kurz vor dem Abschalten ausgeführt.

Jeder Benutzer darf **sync** zu jedem beliebigen Zeitpunkt ausführen; Privilegien des Super-Users werden dafür nicht benötigt.

In Abschnitt 2.7 stellte ich diverse Probleme in Zusammenhang mit verzögertem Schreiben dar, z.B. daß es nicht möglich ist sicherzustellen, daß das Protokoll eines Datenbanksystems physikalisch geschrieben wird, bevor die Datenbank selbst geändert wird. Aus folgenden drei Gründen kann **sync** zur Lösung dieser Probleme nicht verwendet werden:

- **sync** selbst schreibt rein gar nichts — **sync** setzt nur fest, daß geschrieben werden soll. Die Platten-Hardware schreibt die Daten irgendwann, nachdem **sync** zurückgekommen ist.
- **sync** entleert *alle* Puffer, die zur Ausgabe anstehen, nicht nur diejenigen, in die der **sync** aufrufende Prozeß geschrieben hat. Tatsächlich sind die Puffer Eigentum des Kerns, nicht eines Prozesses.
- Es gibt immer noch keine Möglichkeit, Meldungen über physikalische Schreibfehler zu erhalten, wenn ein Puffer entleert wird.

Aus diesen Gründen wird **sync** nur selten von einem Anwendungsprogramm aufgerufen.

9.8 Der Systemaufruf "profil"

```
void profil(buf, nbytes, offset, scale)  /* Zeitprofil erstellen   */
    char * buf;                          /* Zeiger auf Zaehlpuffer */
    int nbytes;                          /* Puffergroesse          */
    int offset;                          /* Programmzaehler-Offset */
    int scale;                           /* Skalierungsfaktor      */
```

profil schaltet ein Zeitprofil eines Prozesses ein, das bei der Erhöhung der Leistung des Prozesses hilft (**profil** findet heraus, wo ein Programm seine Zeit verbringt).

Bei jedem Schlag der Uhr (50, 60 oder 100mal in der Sekunde) wird der Wert des Programmzählers bestimmt und mit diesem ein Wort im Vektor **buf** hochgezählt. Mit Hilfe von **offset** und **scale** wird entschieden, welches Wort hochgezählt werden soll — wir wollen uns jedoch mit diesen Feinheiten nicht zu sehr beschäftigen. Durch einen **scale**-Wert von 0 oder 1 oder durch einen **exec**-Aufruf wird das Zeitprofil abgeschaltet.

Benutzer rufen, falls überhaupt, **profil** selten direkt auf; **profil** wird nur von einer speziellen C-Startfunktion verwendet, die in das Programm eingebunden wird, falls beim **cc**-Kommando die Option **-p** angegeben wird.

9.9 Der Systemaufruf "ptrace"

```
int ptrace(cmd, pid, addr, data)  /* Prozess-Tracing            */
    int cmd;                      /* Kommando                  */
    int pid;                      /* Prozessnummer des Sohns   */
    int addr;                     /* Adresse des Sohns         */
    int data;                     /* zu schreibende Daten      */

/* liefert gelesenen Wert oder -1 bei Fehler                    */
```

ptrace erlaubt einem Vater-Prozeß die Ablaufkontrolle des Sohn-Prozesses mit der Prozeßnummer **pid**. Zum Anschalten der Ablaufkontrolle führt der Sohn **ptrace** mit **cmd** gleich 0 aus. Dies macht man beinahe immer zwischen **fork** und **exec**. Danach hält der Sohn immer genau dann an, wenn er ein Signal empfängt oder einen **exec**-Aufruf ausführt; der Vater erhält dann von **wait** ein Resultat mit einem speziellen Exit-Code. Der Vater kann dann Daten aus dem Adreßraum des Sohns holen, diesen Adreßraum verändern oder den Sohn in Einzelschrittmodus setzen, wodurch der Sohn nach jeder Anweisung ein sogenanntes *Trace-Trap*-Signal (**SIGTRAP**) erhält.

ptrace steht zum alleinigen Gebrauch durch die Debugger **adb** und **sdb** zur Verfügung. Für alle anderen Zwecke ist es viel zu kompliziert; insbesondere sollte **ptrace** nicht als Mittel zur Prozeßkommunikation angesehen werden.

9.10 Der Systemaufruf "times"

```
#include <sys/types.h>  /* nur in System V verfuegbar          */
#include <sys/times.h>

struct tms              /* wird nur fuer System III gebraucht */
{
    time_t tms_utime;   /* Benutzer-CPU-Zeit fuer Prozess      */
    time_t tms_stime;   /* System-CPU-Zeit fuer Prozess        */
    time_t tms_cutime;  /* Benutzer-CPU-Zeit fuer Soehne       */
    time_t tms_cstime;  /* System-CPU-Zeit fuer Soehne         */
};

long times(tbuf)        /* CPU-Zeiten bestimmen                */
    struct tms * tbuf;  /* Zeiger auf Informationspuffer       */

/* liefert benoetigte Real-Time oder -1 bei Fehler             */
```

times meldet vier Zeiten, die für jeden Prozeß automatisch verwaltet werden:

Benutzer-CPU	Zeit zur Ausführung von Anweisungen im Anweisungssegment des Prozesses.
System-CPU	Zeit zur Ausführung von Systemaufrufen im Auftrag des Prozesses.
Benutzer-CPU der Söhne	Summe der Benutzer-CPU-Zeiten aller terminierten Söhne des Prozesses, für die der Vater **wait** aufgerufen hat.
System-CPU der Söhne	Summe der System-CPU-Zeiten aller terminierten Sohn-Prozesse.

Als Resultat wird die seit einem bestimmten Zeitpunkt in der Vergangenheit (i.a. der Zeitpunkt des Systemstarts) verstrichene *Real-Time* (Zeit-auf-der-Uhr-an-der-Wand) geliefert.

Alle Zeiten werden in Einheiten von sogenannten *Ticks* gemessen, wobei ein *Tick* normalerweise eine sechzigstel Sekunde[3] beträgt. Auf manchen Computern jedoch — vor allem bei der 3B-Serie von AT&T — beträgt ein *Tick* eine hundertstel Sekunde. Diese Abweichung macht die Ausgabe von Zeiten unportabel.

In System V wurde die Definitionsdatei **/usr/include/times.h** hinzugefügt, damit Sie die Vereinbarung der Struktur **tms** nicht in Ihr Programm einfügen müssen. Aus Gründen der Portabilität zwischen System III und System V sollten Sie diese

[3] Bei uns 50 Hertz. A.d.Ü.

Definitionsdatei jedoch nicht einfügen. Alternativ dazu können Sie in Ihrem System III auch eine entsprechende Definitionsdatei in den Katalog **/usr/include** eintragen.

Das *Kommando* **time** wird häufig zur Zeitmessung von UNIX-Kommandos benützt. Es ist jedoch einfach, Aufrufe von **times** selbst in ein Programm einzubauen, um die Zeit für verschiedene Programmteile zu messen. Wir haben das in Abschnitt 2.10 gemacht, wo wir zwei Funktionen namens **timestart** und **timestop** benützten. Damit Sie sehen, wie man diese verwendet, zeigen wir Ihnen ein Programm, das 1000mal **getpid** (wahrscheinlich der schnellste aller Systemaufrufe) aufruft:

```
void tmgetpid()  /* Zeit fuer getpid messen */
{
    int i;

    timestart();
    for (i = 0; i < 1000; i++)
        getpid();
    timestop("1000 getpids");
}
```

Dies ist die Ausgabe:

```
1000 getpids: real 1.55; user 0.12; sys 1.43
```

Hier ist nun der Programmtext von **timestart** und **timestop**:

```
#include <sys/types.h>

struct tms
{
    time_t tms_utime;
    time_t tms_stime;
    time_t tms_cutime;
    time_t tms_cstime;
};

long times();
#define TICKS 60.

static struct tms tbuf1;
static long real1;
```

```
void timestart()    /* Uhr starten */
{
    real1 = times(&tbuf1);
}

void timestop(msg)  /* Uhr stoppen */
    char * msg;
{
    struct tms tbuf2;
    long real2;

    real2 = times(&tbuf2);
    fprintf(stderr, "%s: real %.2f; user %.2f; sys %.2f\n", msg,
            (real2 - real1) / TICKS,
            (tbuf2.tms_utime - tbuf1.tms_utime) / TICKS,
            (tbuf2.tms_stime - tbuf1.tms_stime) / TICKS);
}
```

9.11 Der Systemaufruf "time"

```
long time(timep)   /* Systemzeit bestimmen  */
    long * timep;  /* Zeiger auf Zeitangabe */

/* liefert aktuelle Zeit oder -1 bei Fehler */
```

time liefert die Zahl der seit dem 1. Januar 1970 vergangenen Sekunden. In Abschnitt 3.4 wurde ausführlich erklärt, wie UNIX Zeitangaben speichert.

Ist das Argument **timep** nicht **NULL**, so wird die aktuelle Zeit in der long-Integer gespeichert, auf die das Argument zeigt. Dies ist ein Überbleibsel aus der Zeit, als die Sprache C noch keine long-Integer unterstützte. Heutzutage hat man keinen Nutzen davon, da man den Resultatwert mit Hilfe einer einfachen Zuweisung abholen kann. Das Argument von **time** darf immer **NULL** sein.

9.12 Der Systemaufruf "stime"

```
int stime(timep)   /* Systemzeit setzen     */
    long * timep;  /* Zeiger auf Zeitangabe */

/* liefert 0 bei Erfolg oder -1 bei Fehler  */
```

Der Systemaufruf **stime** ist das Gegenstück zu **time**: Er wird (nur durch den Super-User) dazu verwendet, die Systemzeit zu setzen. Normalerweise wird **stime** nur

beim Systemstart und selten direkt aufgerufen, da die Zeit leicht mit dem **date**-*Kommando* gesetzt werden kann (das seinerseits **stime** aufruft). Falls Sie **stime** direkt aufrufen, so muß das Argument in die Zahl der seit dem 1. Januar 1970 vergangenen Sekunden umgewandelt werden, was nicht einfach ist. Die Einzelheiten (und ein Beispielprogramm) finden Sie in Abschnitt 3.4.

9.13 Der Systemaufruf "plock" (nur in System V)

```
#include <sys/lock.h>

int plock(op)  /* Prozess im Speicher fixieren */
    int op;    /* Auswahl                      */

/* liefert 0 bei Erfolg oder -1 bei Fehler     */
```

Normalerweise wird von UNIX ein Prozeß vom Hauptspeicher auf eine Platte ausgelagert, um Platz für einen anderen zu schaffen. Bevor ein ausgelagerter Prozeß ablaufen kann, muß er wieder eingelagert werden. Falls ein ausgelagerter Prozeß durch das Warten auf ein Ereignis blockiert ist, so wird dies die Antwortzeit auf das Ereignis beträchtlich erhöhen. In System V wurde nun der Systemaufruf **plock** hinzugefügt, der einen Prozeß im Speicher festhält, so daß dieser nicht ausgelagert wird. Sobald sich sein Zustand dann von "blockiert" in "bereit" ändert, kann ihn der Systemkern zur Ausführung einplanen.

plock darf nur vom Super-User ausgeführt werden. Das Argument **op** kann einen der folgenden vier Werte annehmen:

PROLOCK	Sperren des gesamten Prozesses, sowohl Anweisungssegment als auch Datensegmente.
TXTLOCK	Sperren des Anweisungs- (Text-) Segments.
DATLOCK	Sperren des Datensegments.
UNLOCK	Aufheben einer beliebigen Sperre.

Eine Sperre bleibt über einen **exec**-Aufruf hinweg erhalten, wird jedoch nicht von Sohn-Prozessen geerbt. Um eine Zersplitterung des Speichers zu vermeiden, sollte man Prozesse, die im Speicher blockiert werden sollen, nach dem Systemstart als erste ausführen. Natürlich können nur wenige Prozesse gesperrt werden, da jeder solche Prozeß den Speicherplatz reduziert, der den normalen Prozessen zur Verfügung steht.

In Xenix 3 und Version 7 gibt es einen ähnlichen Systemaufruf namens **lock**.

9.14 Der Systemaufruf "mount"

```
int mount(spcl, path, rwflag)  /* Dateisystem montieren        */
    char * spcl;               /* Pfad der Block-Geraetedatei */
    char * path;               /* Pfad des Katalogs           */
    int rwflag;                /* Lese-/Schreib-Flagge        */

/* liefert 0 bei Erfolg oder -1 bei Fehler                    */
```

Ein UNIX-Dateisystem (eine logische Platte) kann montiert oder demontiert sein. Ist ein Dateisystem montiert, so wird es Teil der Kataloghierarchie, und die Personen die es benutzen, merken nicht, daß ihre Kataloge sich in Wirklichkeit auf einer anderen (logischen) Platte befinden (abgesehen davon, daß sie mit **link** keinen Verweis über Dateisysteme hinweg anlegen können). Wird eine Platte demontiert, so verschwinden die Kataloge aus der Kataloghierarchie spurlos.

Um **mount** zu verstehen, ist es notwendig, die in Abschnitt 3.3 erklärte Struktur eines Dateisystems zu verstehen. Erinnern Sie sich daran, daß der Wurzelkatalog eines Dateisystems immer die Inumber 2 besitzt. Beim Montieren eines Dateisystems wird der durch **path** gegebene Katalog gleichbedeutend mit dessen Wurzelkatalog. Um diesen Trick durchzuführen, wird die bisher mit **path** verknüpfte Inode im Speicher so verändert, daß alle Verweise, die sich auf diese Inode beziehen, zu Verweisen auf die Inode 2 der montierten (logischen) Platte werden. Von da ab werden Pfade wie gewöhnlich behandelt.

Das erste Argument, **spcl**, ist der Pfadname der Block-Gerätedatei, die montiert werden soll. Diese muß bereits formatiert sein, damit sie das Aussehen eines Dateisystems hat, mit einem Super-Block, Inodes, einer freien Liste usw. Der Kern hat mit dem Formatieren nichts zu tun, da eine Gerätedatei, auch ohne daß sie montiert ist, beliebig gelesen oder beschrieben werden kann. Im Normalfall wird dafür das **mkfs**-Kommando verwendet.

Falls **rwflag**, das dritte Argument, den Wert 1 besitzt, so besteht auf das montierte Dateisystem nur Lesezugriff, ansonsten Lese- und Schreibzugriff. Ein Plattenstapel aus der Datensicherung wird normalerweise nur mit Lesezugriff montiert, damit Benutzer Dateien wiedergewinnen können, ohne den Platteninhalt in Unordnung zu bringen.

mount kann nur vom Super-User ausgeführt werden. Dafür gibt es zwei wichtige Gründe: Erstens kann das fehlerhafte Format eines Dateisystems zum Absturz des Kerns führen. Zweitens kann ein Benutzer, der Schreibzugriff auf die Block-Gerätedatei hat, ein Programm erzeugen, das dem Super-User gehört und bei dem das *set-user-ID*-Bit (siehe Abschnitt 1.6) gesetzt ist — eine Block-Gerätedatei besitzt, solange sie nicht montiert ist, keine besondere Bedeutung; die Blöcke, die die zukünftigen Inodes beherbergen, sind ganz gewöhnliche Datenblöcke. Wird dann das Dateisystem montiert, so wird das *set-user-ID*-Programm Realität.

In der Praxis führt man den **mount**-*Systemaufruf* niemals direkt aus, sondern verwendet das **mount**-*Kommando*. Dies ist wichtig, da beim Montieren eines Dateisystems gewisse Verwaltungstabellen geändert werden sollten, und das **mount**-*Kommando* diese Aufgabe automatisch erledigt.

9.15 Der Systemaufruf "umount"

```
int umount(spcl)  /* Dateisystem demontieren     */
    char * spcl;  /* Pfad der Block-Geraetedatei */

/* liefert 0 bei Erfolg oder -1 bei Fehler       */
```

umount macht die Arbeit von **mount** rückgängig. **umount** prüft, ob alle Kataloge und Dateien geschlossen sind, bevor eine Platte demontiert wird. Wird diese Prüfung nicht erfolgreich bestanden, so mißlingt **umount**. Nach der Ausführung von **umount** befindet sich der Katalog, in dem das Dateisystem montiert war, wieder in seinem ursprünglichen Zustand. Da der Katalog nicht länger mit der Wurzel des demontierten Dateisystems übereinstimmt, verschwinden die darin enthaltenen Verweise auf Kataloge und Dateien in diesem Dateisystem. Nur der Super-User darf **umount** ausführen.

Normalerweise wird das **umount**-*Kommando* zur Ausführung des **umount**-*Systemaufrufs* verwendet, und zwar aus den gleichen Gründen, aus denen das **mount**-*Kommando* zur Ausführung des **mount**-*Systemaufrufs* verwendet wird.

9.16 Der Systemaufruf "acct"

```
int acct(path)    /* Accounting ein- bzw. ausschalten */
    char * path;  /* Pfad der Accounting-Datei        */

/* liefert 0 bei Erfolg oder -1 bei Fehler            */
```

Schaltet man Prozeß-*Accounting* ein, so schreibt der Kern für jeden Prozeß, der zu Ende geht, einen Datensatz mit *Accounting*-Information in die durch **path** gegebene Datei. Dieser enthält den Exit-Status, die Benutzernummer, die Gruppennummer, das Kontrollterminal, die Startzeit, die verbrauchte Gesamtzeit (*Real-Time*), *User-* und *System-Time*, die Speicherbenutzung, die Zahl übertragener Zeichen und Blöcke und den Programmnamen. **NULL** als Wert des Arguments **path** schaltet *Accounting* aus.

Zur Reduzierung der angesammelten *Accounting*-Daten auf sinnvolle Protokolle wird eine große Anzahl von Programmen verwendet. Eines dieser Programme (**accton** in System V) ruft **acct** auf, um *Accounting* an- oder abzuschalten; **acct**

wird selten direkt ausgeführt. Nur der Super-User darf *Accounting* an- oder abschalten.

9.17 Der Systemaufruf "sys3b" (nur in System V)

```
void sys3b(cmd, arg1, arg2)  /* 3B-spezifische Aktion */
    int cmd;                 /* Kommando             */
    int arg1;                /* erstes Argument      */
    int arg2;                /* zweites Argument     */
```

Die 3B-Reihe der AT&T Computer erfordert das Setzen mehrerer Hardwareoptionen, wie z.B. der *maintenance reset function* (MRF) oder der System-Status-Register (SSR). Der Systemaufruf **sys3b**[4] erledigt diese Arbeiten.

Aus zumindest ihnen triftig erscheinenden Gründen haben die Implementierer von UNIX auf den 3B-Computern sich dafür entschieden, diese Optionen mit einem Systemaufruf zu setzen, anstatt einen anderen Weg zu gehen.

Vielleicht wurde kein eigenständiges Programm dafür verwendet, weil man davon ausgeht, daß manche 3B-Computer nie neu gestartet werden, selbst bei Hardware-Wartungsarbeiten. Doch zumindest den Namen hätte man allgemein halten können, z.B. **setop** als Abkürzung für "set option".

Wie dem auch sei, kein gewöhnlicher Benutzer braucht sich um die Verwendung von **sys3b** Sorgen zu machen, und Sie können wetten, daß Sie diesen Systemaufruf auf keinen Computer einer anderen Marke portiert finden.

9.18 Portabilität

Nur wenige der Systemaufrufe in diesem Kapitel sind portabel. **ptrace** ist nicht portabel, da es zum Betrachten von Hardwareregistern und bestimmten Adressen benützt werden kann. **times** ist nicht portabel, da Computer mit verschiedenen Frequenzen getaktet werden. **uname** ist nicht portabel, da seine Ausgabe nicht genau festgelegt ist. **mount** ist nicht portabel, da Namen von Gerätedateien nicht portabel sind. **sys3b** ist nicht portabel, da nicht jeder Computer ein 3B-Computer ist. Und so weiter.

Ein Trost ist, daß dieser Mangel an Portabilität für Anwendungsprogrammierer kein ernsthaftes Problem darstellt, da die betreffenden Systemaufrufe praktisch niemals direkt aufgerufen werden.

[4] Jedes Buch sollte mit einem guten Witz enden.

Aufgaben

9.1 Implementieren Sie **malloc** und **free** (lassen Sie **realloc** weg) und benützen Sie dazu **sbrk**.

9.2 Implementieren Sie das **time**-Kommando.

Anhang A: Prozeßattribute in System V

Dieser Anhang führt alle Prozeßattribute auf und gibt Auskunft darüber, ob ein Attribut über einen **exec**-Aufruf hinweg erhalten bleibt, ob ein Sohn-Prozeß eine Kopie davon erbt und welche Systemaufrufe oder anderen Ereignisse (wie z.B. Termination) das Attribut beeinflussen. Ein Sohn-Prozeß kann ein Attribut mit seinem Vater teilen anstatt eine eigene Kopie zu erhalten; dies ist durch das Wort *gemeinsam* angedeutet. Das Wort *Ausführung* in der mit *beeinflußt durch* bezeichneten Spalte bedeutet, daß das Attribut durch normale Prozeßausführung verändert werden kann, ohne einen Systemaufruf auszuführen.

Attribut	*Bleibt Attribut über* **exec** *hinweg erhalten?*	*Wird Kopie an Söhne vererbt?*	*Attribut wird beeinflußt durch:*
Anweisungssegment	N	J (kann gemeinsam benützt werden)	**exec**
Datensegment	N	J	**exec, brk, sbrk,** Ausführung
Environment (im Datensegment)	optional	J	**exec,** Ausführung
Argumente (im Datensegment)	optional	J	**exec,** Ausführung
wirkliche Benutzernummer	J	J	**setuid**
wirkliche Gruppennummer	J	J	**setgid**
effektive Benutzernummer	möglich	J	**setuid**
effektive Gruppennummer	möglich	J	**setgid**
Prozeßnummer	J	N	**fork**
Prozeßgruppennummer	J	J	**setpgrp**

Attribut	*Bleibt Attribut über* exec *hinweg erhalten?*	*Wird Kopie an Söhne vererbt?*	*Attribut wird beeinflußt durch:*
Vater-Prozeßnummer	J	N	**fork**, Termination des Vaters
Arbeitskatalog	J	J	**chdir**
Wurzelkatalog	J	J	**chroot**
nice-Wert	J	J	**nice**
trace-Flagge	J	N	**ptrace**
Weckuhr	J	J	**alarm**
Maske für Dateianlage	J	J	**umask**
Limitierung der Dateigröße	J	J	**ulimit**
Standardreaktion bei Signalen	J	J	**signal**
ignorierte Signale	J	J	**signal**
abgefangene Signale	N	J	**signal**, Empfang eines Signals
Real Time	J	N	Ausführung
User CPU Time	J	N	Ausführung
System CPU Time	J	N	Ausführung
User CPU Time von Söhnen	J	N	Termination des Sohns
System CPU Time von Söhnen	J	N	Termination des Sohns
Flagge für Ausführungsprofil	N	J	**profil**
Filedeskriptoren	J (N bei *close-on-exec*)	J	**open**, **creat**, **close**, **dup**, **pipe**, **fcntl**
Positionszeiger	J	gemeinsam	**open**, **creat**, **read**, **write**, **lseek**
close-on-exec-Flaggen	J (falls gelöscht)	J	**fcntl**
O_NDELAY-Flaggen	J	gemeinsam	**open**, **fcntl**
Status-Code	—	—	Termination
Prozeß im Speicher fixiert	J	N	**plock**
angeschlossene *Shared-Memory* Segmente	N	J	**shmat**, **shmdt**
Semaphor-Einstellung	J	N	**semop**

Anhang B: Standard-Unterprogramme

Dieser Anhang ist eine kurze Führung durch die Standard-UNIX-Unterprogramme, die in den Beispiel-Programmen dieses Buchs verwendet wurden. Die Beschreibungen sind nur als Gedächtnisstütze gedacht, nicht als maßgebliche Erläuterung. In Kapitel 3 des UNIX-Handbuchs finden Sie eine vollständige Darstellung.

Für viele dieser Funktionen wird die Definitionsdatei **/usr/include/stdio.h** benötigt. Auch wenn eine **#include**-Anweisung dafür nicht explizit aufgeführt ist, können Sie von dem Vorhandensein einer solchen ausgehen.

asctime

```
#include <time.h>

char * asctime(tm)    /* ASCII-Zeit bestimmen */
    struct tm * tm;   /* verstrichene Zeit    */

/* liefert ASCII-Zeit                         */
```

Diese Funktion wandelt die durch **localtime** in einer **tm**-Struktur zur Verfügung gestellte Zeitangabe in eine druckbare ASCII-Zeichenkette um.

atoi

```
int atoi(asc)     /* ASCII in Integer konvertieren */
    char * asc;   /* ASCII-Zahl                    */

/* liefert Integer                                 */
```

Diese Funktion wandelt eine in Form einer ASCII-Zeichenkette angegebene Zahl in eine Integer um.

atol

```
long atol(asc)   /* ASCII in long konvertieren */
    char * asc;  /* ASCII-Zahl                 */

/* liefert long-Integer                       */
```

Diese Funktion wandelt eine in Form einer ASCII-Zeichenkette angegebene Zahl in eine long-Integer um.

fclose

```
int fclose(stream)  /* Dateiverbindung schliessen     */
    FILE * stream;  /* Dateiverbindung                */

/* liefert 0 bei Erfolg oder EOF bei Fehler           */
```

Diese Funktion schließt eine Dateiverbindung und entleert nötigenfalls den Puffer.

fdopen

```
FILE * fdopen(fd, type)  /* FILE-Zeiger mit Filedeskriptor verbinden */
    int fd;              /* Filedeskriptor                           */
    char * type;         /* Art der Ein- bzw. Ausgabe                */

/* liefert einen FILE-Zeiger oder NULL bei Fehler                    */
```

Diese Funktion initialisiert eine Dateiverbindung mit einem bereits aktiven Filedeskriptor. **type** ist entweder `"r"` oder `"w"` (soweit es dieses Buch betrifft).

fflush

```
int fflush(stream)  /* Dateiverbindung entleeren */
    FILE * stream;  /* Dateiverbindung           */

/* liefert 0 bei Erfolg oder EOF bei Fehler      */
```

Diese Funktion entleert eine Dateiverbindung ohne sie zu schließen.

fgets

```
char * fgets(s, n, stream)  /* Zeile lesen                        */
    char * s;               /* Zeichenkette fuer Zeileninhalt     */
    int n;                  /* Groesse von s                      */
    FILE * stream;          /* Dateiverbindung                    */

/* liefert das erste Argument oder NULL bei Fehler oder Dateiende */
```

Diese Funktion liest von einer Dateiverbindung eine Textzeile inklusive des Zeilentrenners und speichert sie in **s**. **s** enthält den Zeilentrenner und ist mit einem Nullbyte abgeschlossen. Es werden höchstens **n**-1 Zeichen gelesen.

fopen

```
FILE * fopen(path, type)  /* Dateiverbindung oeffnen  */
    char * path;          /* Pfadname                 */
    char * type;          /* Art der Ein- und Ausgabe */

/* liefert FILE-Zeiger oder NULL bei Fehler          */
```

Diese Funktion öffnet eine Dateiverbindung zum Lesen oder Schreiben. Soweit es dieses Buch betrifft, ist **type** entweder "r" oder "w".

fprintf

```
int fprintf(stream, fmt, arg, ...) /* formatierte Ausgabe schreiben  */
    FILE * stream;                 /* Dateiverbindung               */
    char * fmt;                    /* Formatbeschreibung            */
    int arg;                       /* Argument (beliebigen Typs)    */

/* liefert die Zahl ausgegebener Zeichen oder negativen Wert bei Fehler */
```

Diese Funktion verhält sich wie **printf**, doch die Ausgabe wandert in die angegebene Dateiverbindung.

fputs

```
int fputs(s, stream)  /* Zeichenkette ausgeben          */
    char * s;         /* Zeichenkette                   */
    FILE * stream;    /* Dateiverbindung                */

/* liefert Zahl ausgegebener Zeichen oder EOF bei Fehler */
```

Diese Funktion schreibt eine Zeichenkette in eine Dateiverbindung, wobei kein Zeilentrenner angehängt wird.

free

```
void free(ptr)   /* Speicherblock freigeben  */
    char * ptr;  /* Zeiger auf Speicherblock */
```

Diese Funktion gibt einen durch **malloc** oder **realloc** angelegten Speicherblock frei.

getc

```
int getc(stream)    /* Zeichen lesen                    */
    FILE * stream;  /* Dateiverbindung                  */

/* liefert Zeichen oder EOF bei Fehler oder Dateiende */
```

Dieser Makro liest und liefert ein Zeichen von einer Dateiverbindung.

getchar

```
int getchar()  /* Zeichen lesen                         */

/* liefert Zeichen oder EOF bei Fehler oder Dateiende */
```

Dieser Makro liest und liefert ein Zeichen vom FILE-Zeiger der Standard-Eingabe (**stdin**).

getenv

```
char * getenv(var)  /* Wert aus dem Environment holen */
    char * var;     /* Variablenname                  */

/* liefert Wert oder NULL, falls keiner vorhanden     */
```

Diese Funktion liefert den Wert der Environment-Variablen, deren Wert in **var** steht.

getgrgid

```
#include <grp.h>

struct group * getgrgid(gid)  /* Eintrag aus Gruppendatei holen */
    int gid;                  /* Gruppennummer                  */

/* liefert Zeiger auf group-Struktur oder NULL bei Fehler       */
```

Diese Funktion durchsucht die Gruppendatei nach dem Eintrag, der der angegebenen Gruppennummer entspricht. Die zurückgelieferte Struktur enthält Teile des Eintrags (Einzelheiten dazu finden Sie im UNIX-Handbuch).

getgrnam

```
#include <grp.h>

struct group * getgrnam(gnm)  /* Eintrag aus Gruppendatei holen */
    char * gnm;               /* Gruppenname                    */

/* liefert Zeiger auf group-Struktur oder NULL bei Fehler      */
```

Diese Funktion durchsucht die Gruppendatei nach dem Eintrag, der dem angegebenen Gruppennamen entspricht. Die zurückgelieferte Struktur enthält Teile des Eintrags (Einzelheiten dazu finden Sie im UNIX-Handbuch).

getpwnam

```
#include <pwd.h>

struct passwd * getpwnam(unm)  /* Eintrag aus Passwortdatei holen */
    char * unm;                /* Login-Name                      */

/* liefert Zeiger auf passwd-Struktur oder NULL bei Fehler      */
```

Diese Funktion durchsucht die Paßwortdatei nach dem Eintrag, der dem angegebenen Login-Namen entspricht. Die zurückgelieferte Struktur enthält Teile des Eintrags (Einzelheiten dazu finden Sie im UNIX-Handbuch).

getpwuid

```
#include <pwd.h>

struct passwd * getpwuid(uid)  /* Eintrag aus Passwortdatei holen */
    int uid;                   /* Benutzernummer                  */

/* liefert Zeiger auf passwd-Struktur oder NULL bei Fehler      */
```

Diese Funktion durchsucht die Paßwortdatei nach dem Eintrag, der der angegebenen Benutzernummer entspricht. Die zurückgelieferte Struktur enthält Teile des Eintrags (Einzelheiten dazu finden Sie im UNIX-Handbuch).

gets

```
char * gets(s)   /* Zeile lesen                              */
    char * s;    /* Zeichenkette zum Aufbewahren der Zeile */

/* liefert Argument oder NULL bei Fehler oder Dateiende    */
```

Diese Funktion liest eine Zeile aus der Standard-Eingabe und legt sie in der Zeichenkette **s** ab. Der Zeilentrenner wird abgeschnitten und die Zeichenkette mit einem Nullbyte abgeschlossen.

localtime

```
#include <time.h>

struct tm * localtime(timep)  /* Zeitangabe in Ortszeit umwandeln */
    long * timep;             /* Zeiger auf Zeitangabe            */

/* liefert Zeiger auf verstrichene Zeit                           */
```

Diese Funktion zerlegt eine Zeitangabe, die als Zahl der seit dem 1. Januar 1970, GMT, vergangenen Sekunden vorliegt, in ihre Bestandteile als Ortszeit um, zur weiteren Verwendung durch **asctime**.

longjmp

```
#include <setjmp.h>

void longjmp(jmpenv, val)  /* an das durch setjmp gesetzte Ziel springen */
    jmp_buf jmpenv;        /* gerettete Umgebung                         */
    int val;               /* Resultatwert fuer setjmp                   */
```

Diese Funktion springt an das durch **setjmp** festgelegte Ziel. Der Stack wird entsprechend aufgeräumt.

malloc

```
char * malloc(size)   /* Speicherblock anlegen          */
    unsigned size;    /* Groesse des anzulegenden Blocks */

/* liefert Zeiger auf Speicherblock oder NULL bei Fehler */
```

Diese Funktion legt einen Block von **size** Bytes dynamisch an und liefert einen Zeiger darauf.

memcpy

```
char * memcpy(to, from, nbytes)  /* Daten kopieren                    */
    char * to;                   /* Ziel                             */
    char * from;                 /* Quelle                           */
    int nbytes;                  /* Zahl der zu kopierenden Bytes */

/* liefert erstes Argument                                           */
```

Diese Funktion — die es nur in System V gibt — kopiert Daten nackt, ohne Rücksicht auf Nullbytes.

printf

```
int printf(fmt, arg, ...)  /* formatierte Ausgabe schreiben          */
    char * fmt;            /* Formatbeschreibung                     */
    int arg;               /* Argument (beliebigen Typs)             */

/* liefert die Zahl ausgegebener Zeichen oder negativen Wert bei Fehler */
```

Diese Funktion formatiert ihre Argumente anhand der durch **fmt** gegebenen Formatbeschreibung und gibt die resultierende Zeichenkette auf die Standard-Ausgabe aus.

putc

```
int putc(c, stream)   /* Zeichen ausgeben */
    char c;           /* Zeichen          */
    FILE * stream;    /* Dateiverbindung  */

/* liefert Zeichen oder EOF bei Fehler    */
```

Dieser Makro schreibt ein Zeichen in eine Dateiverbindung.

realloc

```
char * realloc(ptr, size)  /* Speicherblock aendern                 */
    char * ptr;            /* Zeiger auf alten Speicherblock        */
    unsigned size;         /* Groesse des anzulegenden Blocks       */

/* liefert Zeiger auf geaenderten Speicherblock oder NULL bei Fehler */
```

Diese Funktion prüft die Größe eines bestehenden Speicherblocks, um sicherzustellen, daß er mindestens **size** Bytes umfaßt. Falls nötig, wird der Block vergrößert oder ein neuer Block angelegt und die alten Daten dorthin kopiert.

setbuf

```
void setbuf(stream, buf)  /* Pufferung steuern  */
    FILE * stream;        /* Dateiverbindung    */
    char * buf;           /* Zeiger oder NULL   */
```

Diese Funktion verbindet einen Puffer mit einer Dateiverbindung oder schaltet die Pufferung aus, falls das zweite Argument **NULL** ist.

setjmp

```
#include <setjmp.h>

int setjmp(jmpenv)      /* Ziel fuer longjmp setzen   */
    jmp_buf jmpenv;     /* gerettete Umgebung         */

/* liefert 0 oder den durch longjmp gelieferten Wert */
```

Diese Funktion speichert im Puffer **jmpenv** eine Zielposition für **longjmp**. Wird ein entsprechender **longjmp**-Aufruf ausgeführt, so scheint **setjmp** mit dem von **longjmp** gelieferten Wert zurückzukehren.

sleep

```
unsigned sleep(secs)  /* schlafen gehen */
    unsigned secs;    /* Sekundenzahl   */

/* liefert die nichtverschlafene Zeit   */
```

Diese Funktion benützt die Systemaufrufe **alarm** und **pause**, um eine bestimmte Zahl von Sekunden zu schlafen.

sprintf

```
int sprintf(s, fmt, arg, ...)    /* formatierte Ausgabe schreiben      */
    char * s;                    /* Zeichenkette                       */
    char * fmt;                  /* Formatbeschreibung                 */
    int arg;                     /* Argument (beliebigen Typs)         */

/* liefert Zahl resultierender Zeichen oder negativen Wert bei Fehler   */
```

Diese Funktion verhält sich wie **printf**, jedoch wandert die Ausgabe in die Zeichenkette **s**.

sscanf

```
int sscanf(s, fmt, varp, ...)  /* Zeichenkette gemaess Format durchsuchen */
    char * s;                  /* Zeichenkette                            */
    char * fmt;                /* Format                                  */
    int * varp;                /* Zeiger auf Variable                     */

/* liefert Zahl umgewandelter Objekte oder EOF bei Fehler                 */
```

Diese Funktion durchsucht die Zeichenkette **s** gemäß der durch **fmt** gegebenen Formatbeschreibung. Objekte werden umgewandelt und den Variablen in der angetroffenen Reihenfolge zugewiesen.

strcat

```
char * strcat(to, from)  /* Zeichenketten verketten */
    char * to;           /* Ziel                    */
    char * from;         /* Quelle                  */

/* liefert erstes Argument                         */
```

Diese Funktion kopiert das zweite Argument an das Ende des ersten Arguments.

strchr

```
char * strchr(s, c)  /* Zeichen in Zeichenkette finden          */
    char * s;        /* Zeichenkette                            */
    char c;          /* Zeichen                                 */

/* liefert Zeiger auf die Stelle, an der das Zeichen c das erste Mal */
/* vorkommt, oder NULL, falls das Zeichen c in s nicht vorhanden ist */
```

Diese Funktion durchsucht eine Zeichenkette nach dem ersten Auftreten eines angegebenen Zeichens.

strcmp

```
int strcmp(s1, s2)  /* Zeichenketten vergleichen */
    char * s1;      /* Zeichenkette              */
    char * s2;      /* Zeichenkette              */

/* liefert Integer, die das Verhaeltnis angibt   */
```

Diese Funktion vergleicht zwei Zeichenketten lexikographisch; sie liefert eine negative Zahl falls **s1** kleiner ist als **s2**, Null, falls sie gleich sind, und eine positive Zahl, falls **s1** größer ist als **s2**.

strcpy

```
char * strcpy(to, from)  /* Zeichenketten kopieren */
    char * to;           /* Ziel                   */
    char * from;         /* Quelle                 */

/* liefert erstes Argument                         */
```

Diese Funktion kopiert die mit einem Nullbyte abgeschlossene Zeichenkette **from** an die Stelle, auf die **to** zeigt.

strcspn

```
int strcspn(s, set)  /* Vorsilbe finden   */
    char * s;        /* Zeichenkette      */
    char * set;      /* Menge von Zeichen */

/* liefert Laenge der Vorsilbe            */
```

Diese Funktion liefert die Länge der längsten Vorsilbe von **s**, die nur aus Zeichen aus **set** besteht.

strlen

```
int strlen(s)  /* Laenge der Zeichenkette bestimmen */
    char * s;  /* Zeichenkette                      */

/* liefert Laenge                                   */
```

Diese Funktion liefert die Länge einer Zeichenkette (ohne das abschließende Nullbyte).

strncpy

```
char * strncpy(to, from, n)  /* n Zeichen der Zeichenkette kopieren */
    char * to;               /* Ziel                                */
    char * from;             /* Quelle                              */
    int n;                   /* Zahl der Zeichen                    */

/* liefert erstes Argument                                          */
```

Diese Funktion kopiert maximal **n** Zeichen der mit einem Nullbyte abgeschlossenen Zeichenkette **from** an die Stelle, auf die **to** zeigt. Der Kopiervorgang wird abgebrochen, sobald ein Nullbyte entdeckt wird. Das Ziel ist nicht notwendigerweise mit einem Nullbyte abgeschlossen.

strrchr

```
char * strrchr(s, c)  /* Zeichen in Zeichenkette finden          */
    char * s;         /* Zeichenkette                            */
    char c;           /* Zeichen                                 */

/* liefert Zeiger auf die Stelle, an der das Zeichen das letzte Mal */
/* vorkommt, oder NULL, falls das Zeichen nicht vorhanden ist       */
```

Diese Funktion durchsucht eine Zeichenkette nach dem letzten Auftreten eines angegebenen Zeichens.

strtok

```
char * strtok(subj, delim)  /* Zeichenkette in Abschnitte aufspalten  */
    char * subj;            /* Zeichenkette                           */
    char * delim;           /* Begrenzer                              */

/* liefert Zeiger auf subj oder NULL, falls kein Abschnitt uebrigbleibt */
```

Diese Funktion liefert bei jedem Aufruf einen Abschnitt von **subj**. Abschnitte werden durch ein in **delim** angegebenes Zeichen getrennt. Der erste Aufruf gibt die zu durchsuchende Zeichenkette an; in diese werden während des Auffindens der Abschnitte Nullbytes kopiert, um diese abzuschließen. Bei allen weiteren Aufrufen für die gleiche Zeichenkette muß das erste Argument **NULL** sein.

system

```
int system(cmd)   /* Shell-Kommandozeile ausfuehren */
    char * cmd;   /* Kommando                       */

/* liefert Exit-Code der Shell                     */
```

Diese Funktion erzeugt einen Sohn-Prozeß, der die Standard-Shell **sh** aufruft und ihr die angegebene Kommandozeile zur Ausführung übergibt.

ungetc

```
int ungetc(c, stream)  /* Zeichen in Eingabe zurueckstellen */
    char c;            /* Zeichen                           */
    FILE * stream;     /* Dateiverbindung                   */

/* liefert Zeichen oder EOF bei Fehler                      */
```

Diese Funktion stellt ein Zeichen in eine Dateiverbindung zur Eingabe zurück, so daß es dort später gelesen werden kann.

Literaturverzeichnis

Bücher, auf die im Text direkt Bezug genommen wird, sind dort in Fußnoten aufgeführt. Dieses Literaturverzeichnis beinhaltet eine Auswahl an Büchern, die dem Leser bei der Vorbereitung auf das vorliegende Werk helfen oder Gelegenheiten für tiefergehendere Betrachtungen bieten.[1]

Vorbereitende Bücher

Bell Laboratories.

The Bell System Technical Journal 57, no. 6, pt. 2 (July/August 1978). Dieses Werk enthält über 400 Seiten mit Artikeln, die von vielen Entwicklern von UNIX, von UNIX-Kommandos und Anwendungen, die auf UNIX basieren, stammen. Es stellt einen ausgezeichneten Hintergrund über die Entwicklungen der frühen und mittleren 70er Jahre dar.

Eine weitere Ausgabe, Oktober 1984, befaßt sich ebenfalls nur mit UNIX. (A.d.Ü.)

Bourne, S. R.

The UNIX System. Reading, Mass.: Addison-Wesley Publishing Company, Inc., 1982. Eine vermittelnde Einführung in UNIX. Dieses Buch befaßt sich ein wenig mit Editieren (mittels **ed** und **vi**), ein bißchen mit der Shell (die von Bourne entwickelt wurde), ein wenig mit Programmieren und ein wenig mit **nroff**.

Deutsch: *Das UNIX System*, Addison-Wesley Verlag (Deutschland) GmbH, 1985. (A.d.Ü.)

[1] Das Literaturverzeichnis wurde für die deutsche Ausgabe aktualisiert und erweitert. A.d.Ü.

Comer, Douglas.
Operating System Design: The XINU Approach. Englewood Cliffs, N.J.: Prentice Hall, Inc., 1984. Ein genaue Betrachtung des Innenlebens eines Betriebssystems, das ein wenig UNIX gleicht. ("XINU" steht für "XINU is not UNIX.") Das Buch enthält einen Ausdruck des ganzen Quell-Codes.

Lomuto, Ann Nicols, und Lomuto, Nico.
A UNIX Primer. Englewood Cliffs, N.J.: Prentice Hall, Inc., 1983. Dieses Buch stellt für Anfänger eine gute Einführung in UNIX dar. Es gibt Dutzende von einführenden Büchern — darunter ist dieses eines der Besten.

Peterson, James L., und Silberschatz, Abraham.
Operating System Concepts. Reading, Mass.: Addison-Wesley Publishing Company, Inc., 1983. Eine gute Einführung in die Prinzipien von Betriebssystemen, darunter auch die meisten von UNIX.

Bücher über C und UNIX-Programmierung

Kernighan, Brian W., und Pike, Rob.
The UNIX Programming Environment. Englewood Cliffs, N.J.: Prentice Hall, Inc., 1984. Eine ausgezeichnete Einführung in die UNIX-Programmierung unter Verwendung der Shell, von C und verwandten Werkzeugen wie **make** und **yacc**. Dieses Buch stellt die bestmögliche Vorbereitung auf das vorliegende Buch dar.

Deutsch: *Der UNIX Werkzeugkasten*, Carl Hanser Verlag, 1987. (A.d.Ü.)

Kernighan, Brian W., und Ritchie, Dennis M.
The C Programming Language. Englewood Cliffs, N.J.: Prentice Hall, Inc., 1978. Das erste — und noch immer das beste — Lehrbuch über C. Anhang A ist das *C Reference Manual*, geschrieben von Dennis M. Ritchie, dem Erfinder von C.

Deutsch: *Programmieren in C*, Carl Hanser Verlag, 1983. (1988 wird eine neue Ausgabe erscheinen.) (A.d.Ü.)

Plum, Thomas.
Learning to Program in C. Cardiff, N.J.: Prentice Hall, Inc., 1983. Ein weiteres Lehrbuch über C. Manche Leute finden, daß man diesem Buch leichter folgen kann als *The C Programming Language.*

Deutsch: *Das C-Lehrbuch*, Carl Hanser Verlag, 1985. (A.d.Ü.)

UNIX-Handbücher

AT&T.
AT&T UNIX System V User's Reference Manual. Prentice-Hall Inc., Englewood Cliffs, N.J., 1987. (A.d.Ü.)

AT&T.
AT&T UNIX System V Programmer's Reference Manual. Prentice-Hall Inc., Englewood Cliffs, N.J., 1987. (A.d.Ü.)

Bell Laboratories.
UNIX Programmer's Manual, Vol. 1, 2. New York: Holt, Rinehart and Winston, 1979. Dieses Buch ist das meistverfügbarste veröffentlichte UNIX-Handbuch, doch bezieht es sich auf Version 7 und ist für System III, System V oder Xenix 3 ungeeignet. *Caveat emptor.*

Bell Laboratories.
UNIX System Administrator's Manual, System V. Western Electric Company, Inc., 1983. Der Teil des Handbuchs für den Systemadministrator.

Bell Laboratories.
UNIX System User's Manual, System V. Western Electric Company, Inc., 1983. Falls Sie kein UNIX-Benutzer sind, sollten Sie sich dieses UNIX-Handbuch besorgen. Anderenfalls wird Ihnen das Handbuch, das mit Ihrem System mitgeliefert wird, besser dienen.

D.E. Kevorkian
System V Interface Definition, Issue 2. At&T Customer Information Center, IN, 1986. (A.d.Ü.)

/usr/group.
Proposed Standard. Santa Clara, Calif.: */usr/group*, 1984. Der UNIX-Standard, der von einem Komitee der industriellen Vereinigung */usr/group* vorgeschlagen worden ist. Er stellt im wesentlichen eine Untermenge von System III dar.

/usr/group.
Reviewer's Guide to the Proposed /usr/group Standard. Santa Clara, Calif.: */usr/group*, 1984. Dieser Führer enthält Kommentare zu den Überlegungen, die hinter dem *Proposed Standard* stehen, speziell, weshalb gewisse Punkte aufgenommen oder ausgelassen wurden.

X/OPEN.
X/OPEN Portability Guide, Issue 2, Volume 1-5. Elsevier Science Publishers B.V., Amsterdam, 1987. (A.d.Ü.)

Stichwortverzeichnis

.profile, Login-Kommandodatei 293
/bin/sh, Kommando 132
/dev, Katalog 56, 74
/dev/null, Gerätedatei 181
/dev/tty, Terminal-Gerätedatei 94, 97
/etc, Katalog 71
/etc/group *9*, 82, 313
/etc/passwd *9*, 71, 82, 147, 149, 313
/tmp, Katalog 29, 71, 165
/usr/bin, Katalog 161
/usr/group 50, 98, 114, 198, 200, 258
/usr/include, Katalog 300
/usr/include/errno.h, Def.datei 19, 29, 38
/usr/include/fcntl.h, Def.datei 19, 31, 88
/usr/include/signal.h, Def.datei 264
/usr/include/stdio.h, Def.datei 309
/usr/include/sys/dir.h, Def.datei 54
/usr/include/sys/filsys.h, Def.datei 60
/usr/include/sys/ino.h, Def.datei 60
/usr/include/sys/stat.h, Def.datei 67, 78
/usr/include/sys/types.h, Def.datei 78
/usr/include/termio.h, Def.datei 104
/usr/include/times.h, Def.datei 299
., Katalog 55, 69
.., Katalog 55, 69, 150
/, Katalog 60, 149
Abbruch durch Signale *266*, 267
Abfangen von Signalen 7, 143, 269–270, 276, 279, 285
Ablaufverfolgung eines Prozesses 265, 304
abort, Standardfunktion 265
absoluter Pfad 3, 60
access, Systemaufruf 71–73, 91
Accounting 304
accton, Kommando 304
adb, Debugger 12, 264, 298
addmsg.h, Definitionsdatei 211
Adoptionsbüro 55
Adreßraum 6, 12, 15, 266, 298
aktive Filedeskriptoren 6, 78, 128
aktive Inodes 78
aktueller Katalog 3, 6
alarm, Systemaufruf 33, 264, 267, *279*, 280, 285–286, 316
Alarm-Clock 267
Anweisungssegment 6, 68, 126–127, 138, 143, 299, 302
  initialisieren 141
  Lesezugriff 6, 141
Arbeitskatalog 25, 91, 128, 148
  ändern 149
Argument eines Programms 117
Argumente in Anführungszeichen, Shell 155
ASCII-Zeichensatz 106
asctime, Standardfunktion 62, 82, 309, 314
Assembler 5
Assemblerprogramm 15
Assembliere 5
assign, Funktion *121*, 122
  3B-Computer 299, 305
    Systemaufruf sys3b 305
  UNIX-Version 21, 32, 50, 93, 186, 286
atoi, Standardfunktion 160, 309
atol, Standardfunktion 310
atomar
  open 34
  Systemaufrufe 35
Aufhören, bei Signalen 265
ausführbare Datei 5–6, 132
Ausführungsrecht 26, 72, 149
Ausführungszeit 62, 128, 139
Ausgabe
  auf Terminal 97–98, 129
    Zeilenpufferung 129
  für gewöhnliche Dateien 23
  in Datei 129
  in Pipe 129
  ungepufferte 129
Ausgabe umlenken 167
Auslagern eines Prozesses 57
Austauschtechniken, vorsichtige 37, 51
auto, Speicherklasse 18
Bücherei 5, 213
  Signale 265
B-Baum 51
Band 77
Baud-Rate 104
Baum 51

vollkommener 69, 71
bc, Kommando 195
Bell Laboratorien 197, 201–202, 250
Bell-System 197
Benutzer
Identität 9
Zugriffsrechte 9
Benutzer-CPU Zeit 299
Benutzer-Datensegment 6, 126–127, 129, 138, 143
maximale Größe 289
Benutzer-Prozeß 14
Benutzernummer 2, *9*, 76, 79, 304, 313
effektive 9, 30, 56, 69, 76, 127–128, 147, 278
in einer Inode 10
wirkliche 9, 71–72, 148, 278
Benutzerprozeß 5, 40
Benutzerpufferung 40, 41, 45–46
Paket 41
Benutzerschnittstelle 115
bereit, Prozeßzustand 302
Berkeley 4.2 BSD 21, 90, 280
Blockgröße 40
curses-Paket 103
Dateiattribute 241
Messages 262
Shared-Memory 262
Signale 264, 286
Systemaufrufe
dup2 193
execlp 152
execvp 152
fcntl 90
mkdir 90
open 50
sigvec 286
utimes 90
vfork 141, 153
Terminal-Gerätetreiber 113
Berkeley-Shell, csh 141, 194
Betriebsmittel 139
bidirektionale Pipes 156
Bildschirmeditor 20, 94, 103, 109, 115, 188
Bildschirmspeicher 114
Bildschirmterminal 114
Bildschirmzugriff 114
Binder 5–6
Binder, ld 291
bit-mapped-Bildschirm 93
Blöcke
Liste der freien 60
Block "0", Dateisystem 60
Block "1", Dateisystem 60
Block "2", Dateisystem 60
Block-Gerätedateien 4, 40, 57, 67, 303
Block-System der Ein- und Ausgabe 56
Blockdepot 4, 36–37, 56–57, 59–60, 297
bei Pipes 157
Blockgröße 40
Blockieren einer Pipe 156
blockiert, Prozeßzustand 302
Blockierung bei Terminaleingabe 98
Blockierung von Systemaufrufen 279
blockierungsfreie Ein- und Ausgabe für Terminals 98, 280
blocking 98
Blockmodell 41, 57
Änderungszeitstempel 11, 79
break 105
break-Wert 291
brk, Systemaufruf 244–245, 290, *291*, 292
BRKINT-Bit 111
buffer cache 4
builtin, Funktion 179, 182
Bus-Error 266
busy-wait
bei einem DBMS 250
bei Semaphoren 235
C 1, 15, *20*, 41, 48
Bücher 1
Bücherei 29, 142
C-Programm 117
Compiler 48
Option -f 265
Option -p 298
Handbuch 18
Nachschlagewerk 1
Operatoren 18
Portabilität von Programmen 20
Präprozessor 90
Schnittstelle zu Systemaufrufen 132
Startfunktion 298
Systemaufrufe 15
Unterprogramme 15
Werterückgabe durch Funktionsargument 122
Zuweisung 301
canonical, kanonische Eingabe 109
cat, Kommando 164–165
catchsig, Funktion 274
cd, Kommando 149, 167, 182, 293
cget, Funktion *101*, 103
Character-Gerätedatei 4, 57, 67
chdir, Systemaufruf 149
chmod, Kommando 11
chmod, Systemaufruf 26, 67, 76, 79, 88, 292
chown, Kommando 11
chown, Systemaufruf 76, 79
chroot, Systemaufruf 3, *149*, 150, 152

chsize, Systemaufruf 2
cleanup, Funktion 274
client 198
close, Systemaufruf 23–24, 30, ***39***, 94, 98, 113, 129, 157, 163
close-on-exec-Flagge 88–90, 130
COFF, Common Object File Format 5
Columbus UNIX 197
command, Funktion ***173***, 174–175, 177–178
Compiler 5, 18
Context-Switch 15
contiguous 68
control terminal 8
Control-\ 265
Control-D, Dateiende 95, 98, 104, 146, 168
Control-J 112
Control-Q 107
Control-S 107
Control-X 98
copy-on-write-Verfahren 141
core dump, Speicherauszug 143, 265
cp, Kommando 23
CPU 59
CPU-Zeit 6
  angesammelte 6
cready, Funktion ***101***, 103
creat, Systemaufruf 23–24, ***25***, 27, 29–33, 35, 48, 53, 56, 67, 74, 89, 95, 157, 235, 241, 292–293
  atomar 30
  Version 7 30
creatsem, Systemaufruf 241
cron, Hintergrund-Prozeß 297
csh, Berkeley-Shell 141, 194
ctime, Standardpaket 62–63, 66
current directory 3
curses-Paket 93, 103, 116, 195
date, Kommando 302
Datei 1, 31, 56, 304
  Änderungszeitstempel 11
  Attribute 66
  ausführbare 5–6, 132
  Ausführungsrecht 66
  Besitzer ***26***, 57
  Block-Gerätedatei 57
  Ein- und Ausgabe 271
  FIFO-Datei 1, 4, 14, 67, 200, ***201***, 205, 231, 233
    E/A-Systemaufrufe 201–202
    Ersatz für Pipe 202
    Implementierung von Messages 202
    in System III 197
  Gerätedatei 1
  gewöhnliche 1, 3, 5, 10, 53, 55, 57, 74, 90
  Katalog 1, 10
  Leserecht 24, 66
  maximale Größe 289
  nackte Platten-Gerätedatei 59
  neue 27
  Objektdatei 5
  Platten-Gerätedatei 59
  Prozeßkommunikation 13
  Rückwärtslesen 41
  Rechte 24, 26, 66
  Schreibrecht 24, 66
  Semaphor 13, 27, 30, 33, 34
  Special-File 1, 4, 10
  Statusinformation 53
  temporäre 26, 28
  Textdatei 5
  Zeitstempel 26, 57, 62
    letzte Änderung 26
    letzte Statusänderung 26
    letzter Zugriff 11, 26
  Zugriffsschutz 66, 199
  Zugriffszeitstempel 11, 26
Dateiart 60
Dateiattribute ***66***, 72, 76, 79
Dateiende 24, 34, 51, 95, 97–98, 104, 135, 146, 156–157, 165, 168, 172, 181, 186, 189, 190, 194
  Position hinter 2
Dateigröße 2, 37
Dateisystem 2–3, 36, 56, 91, 303
  Block "0" 60
  Block "1" 60
  Block "2" 60
  demontieren 303–304
  erzeugen 303
  freie Liste 303
  Implementierung 1
  Inode 303
    "1" 60
  Kennsatz 294
  Lesezugriff 303
  logische Platte 294
  montieren 294, 303
  reparieren 2
  Schreibzugriff 303
  Super-Block 294, 303
  Verweis 303
  Wurzelkatalog 149–150, 303–304
Dateityp 67, 90
Dateiverbindung
  öffnen 311
  entleeren 310
  initialisieren 310
  mit Puffer verbinden 316
  Pufferung ausschalten 316

schließen 310
Zeichen zur Eingabe zurückstecken 320
Datenübertragung 4, 36
Datenübertragungsleitung 77
Datenübertragungsvorrechner 93
Datenbank
Deadlock-Situation 257
Konsistenz 256
Protokoll zum Zurücksetzen 256
Serialisierbarkeit 256
Sperre 256
exklusive 256
gemeinsame 256
Suche nach einem Opfer 257
Transaktion 256
Datenbank-Managementsystem, DBMS 57, 198
Datenbank-Anwendungen 4
Datenbank-Manager 4, 59
Datenbank-Managerprozeß 58
Datenbasis, bei DBMS 198
Datenblöcke 60, 303
Datenerfassungs-Systeme 57
Datensegment 57, 291, 302
gemeinsam benützen 141
initialisieren 141
maximale Größe ändern 290
vergrößern 292
Datensichtgerät 93
Datentyp
enum 17
void 17
DATLOCK-Bit 302
DBMS 198, 250
busy-wait 250
Datenbasis 198
Deadlock-Situation 242
dezentralisiertes 199
in Version 7 261
Klient 198
Lock-Manager 199, 259, 262
Satzsperren 242
zentralisiertes 199, 259
Zugriffsroutinen 198
dbms.h, Definitionsdatei *214*, 218
dbmscall, Funktion 222
dbmstest, Programm 226
dc, Kommando 195
Dclose, Funktion 214
Dcreate, Funktion 214
Ddelete, Funktion 217
Deadlock-Situation 189, 193
bei bidirektionalen Pipes 156
bei DBMS 242
bei Pipes 158, 186
bei Transaktionen 257
bei zwei Pipes 188
in System V 242
in Xenix 3 242
Debugger 13
adb 12, 264, 298
sdb 12, 264, 298
DEC
VAX-Computer 141
VMS Betriebssystem 153
Definitionsdatei
/usr/include/errno.h 19, 29, 38
/usr/include/fcntl.h 19, 31, 88
/usr/include/signal.h 264
/usr/include/stdio.h 309
/usr/include/sys/dir.h 54
/usr/include/sys/filsys.h 60
/usr/include/sys/ino.h 60
/usr/include/sys/stat.h 67, 78
/usr/include/sys/types.h 78
/usr/include/termio.h 104
/usr/include/times.h 299
addmsg.h 211
dbms.h 214, 218
defs.h 19
dir.h 54
ioctl.h 113
message.h 209
sgtty.h 113
stream.h 41
vararg.h 152
DEL 7
device number 5
dezentralisiertes DBMS 199
Dget, Funktion *214*, 215–216
Dgetnext, Funktion 216
Diagnose-Ausgabe 25, 94
dir.h, Definitionsdatei 54
direct memory access, DMA 4, 57, 59
directory 1
Division durch "0" 266
DMA, direct memory access 4, 57, 59
Dopen, Funktion 214
Dput, Funktion 216
Drohnen-Prozeß 58
Drucker-Spooler 13
Dtop, Funktion 215
dup, Systemaufruf 25, 88–89, 157, *162*, 163–165, 181, 193–194
dup2, Systemaufruf 89
dup2, in Berkeley 4.2 BSD 193
dup2, in Version 7 193
Duplex-Betrieb, bei Terminals 108
dynamische Speicherverwaltung 286, 291, 314
EACCESS, Fehler-Code 29, 259

EAGAIN, Fehler-Code 98
ECHILD, Fehler-Code 143, 274
ECHO-Bit 108, 111–112
ECHOE-Bit 108
ECHOK-Bit 108
ed, Editor 1, 13, 27, 165, 188, 271, 291
  Signal 275
EDEADLOCK, Fehler-Code 259
edinvoke, Funktion 190
Editor
  Bildschirmeditor 94, 109, 115, 188
  Bildschirmeditor EDIX 114
  ed 27, 165, 188, 291
  vi 103, 280–281, 286
EDIX, Bildschirmeditor in PC/IX 114
edrcv, Funktion 191
edsnd, Funktion 191
effektive Benutzernummer *9*, 26, 30, 56, 69, 76, 127–128, 147, 278
  "0" 55
effektive Gruppennummer *9*, 26, 30, 69, 76, 128, 147
EFBIG, Fehler-Code 38
effective group-ID 9
effective user-ID 9
Ein- und Ausgabe
  Block-Gerät 56
  Block-Gerätedatei 56
  Block-System 56
  für Dateien 271
  für Kataloge 53
  für Platten-Gerätedateien 53
  für Terminals 93–94
  gepufferte 51
  gewöhnliche Dateien 59
  nackte Platten-Gerätedateien 59
  physische 60
  Platten-Gerätedateien 59
  schnellere 57
  Systemaufrufe 53
  Umlenkung 94
  Unterschiede 59
Ein- und Ausgabeumlenkung 155
Ein- und Auslagern von Prozessen 68, 139, 302
Einbruch 76
einfaches Kommando 167, 173
Eingabe
  blockierungsfreie 286
  blockweise 97
  für gewöhnliche Dateien 23
  kanonische 109
  puffernde 97
  verzögerungsfreie 109, 112
  vom Terminal 286
  Wartezustand 103
  zeilenweise 97
Eingabe umlenken 167
Eingabe-Warteschlange 105, 107, 109
  entleeren 105
Eingabeflußsteuerung 107
eingebaute Kommandos in der Shell 167
EINTR, Fehler-Code 271–272, 279
EINVAL, Fehler-Code 45, 50
Einzelschrittmodus 298
EIO, Fehler-Code 58
EMT, PDP-11 Maschinenbefehl 265
Emulator-Trap-Instruction 266
ENAVAIL, Fehler-Code in Xenix 3 241
end, Variable 291
end-of-file, Dateiende 156–157, 165, 181, 186, 189, 190, 194, 266
end-of-transmission, Übertragungsende 95
endliche Maschine 168, 281–282
  Zustand 168
ENOENT, Fehler-Code 70
ENOMEM, Fehler-Code 292
entrysig, Funktion 180, 269
enum, Datentyp 17
Environment *117*, 119, 124, 126, 128, 134–135, 138, 152
  ändern 117, 119
  ausgeben 118, 134–135, 138
  erzeugen 119
  Suche via PATH 131
  Symboltabelle 119
  Übergabe 131
  Variable 118, 312
    exportieren 134–135
    HOME 182
    PATH 130, 132, 134, 194
    PS2 136
    TZ 62–63, 66, 118
    Vererbung 120
  Zeichenketten 118
envprint, Programm 118, 138
EOL 109
  am Zeilenanfang 95
EOT, Dateiende 97, 98, 104, 109, 112, 115, 146, 168, 172
Erase, 95, 98, 104
errno, externe Variable 16, 29, 33, 37, 45, 58, 68, 98, 143, 165, 218, 241, 272, 279
Escape-Sequenz 115, *280*, 281
Escape-Zeichen 281, 285
ETXTBSY-Bit 68
EVexport, Funktion 122
EVget, Funktion 123
EVinit, Funktion 123

EVprint, Funktion 125
EVset, Funktion 121
EVupdate, Funktion *124*, 126
exec, Systemaufruf 88, 117, *126*, 127, 130, 134, 138–139, 141, 148, 153, 161–163, 178, 181, 269, 279, 298, 302
  execl *127*, 128–131, 132, 161
  execle 131
  execlp *131*, 132, 152, 161
  execv *131*, 132
  execve 131
  execvp *131*, 132, 137–138, 140–141, 151–152, 161, 173
  Signalreaktionen 269
execlp2, Funktion 152
execut2, Funktion 140, 145
execut3, Funktion 182
execute, Funktion 140
execvp2, Funktion 151–152
exit, Systemaufruf 16, *142*, 143, 181, 267, 269, 274
Exit-Code, von Prozessen 181, 269, 274, 298, 304
export, eingebautes Kommando 119, 125, 134–135, 152, 167
Füllgrad einer Pipe 158
fflush, Standardfunktion 310
find, Funktion *120*, 121
find, Kommando 69
Flußsteuerung 107
F_DUPFD-Bit 163
fatal, Funktion 17, 183, 277
fclose, Standardfunktion 142, 310
fcntl, Systemaufruf 25, *88*, 89–90, 94, 98–99, 104, 130, 156–157, 163, 193–194, 200
  in Version 7 163
fdopen, Standardfunktion 98, 190, 310
Fehler bei Gleitkommarechnung 266
Fehler-Code 16, *17*, 45
  EACCESS 259
  EAGAIN 98
  ECHILD 143, 274
  EDEADLOCK 259
  EFBIG 38
  EINTR 271–272, 279
  ENAVAIL 241
  ENOMEM 292
  EACCES 29
  EINVAL 45, 50
  EIO 58
  ENOENT 70
  errno 58, 68
Fernschreiber 93
fgets, Standardfunktion 310
FIFO-Datei 1, 4, 14, 20, 67, 74–75, 78, 90, 155, 200, *201*, 205, 231, 233
  Dateiattribute 67
  Ein- und Ausgabe 23, 201-202
  Ersatz für Pipe 202
  Implementierung von Messages 202
  in System III 197, 202, 296
file mode creation mask 26, 292
file size limit 12
File-Locking 2, 5, 200
FILE-Zeiger von stdio 98
Filedeskriptor 6, 12, 17, *24*, 29–30, 36, 39, 78, 88, 100, 139, 190, 205, 310
  aktiver 6, 78, 139, 163
  bei Pipes 156
  Duplikat 163
  gemeinsam benützter 162
  kleinster verfügbarer 30
  schließen 98, 129, 142
  mit Terminal verbinden 94, 98
  verdoppeln 162
  vererben 159
  verfügbar machen 163
  "0" *25*, 162–163
  "1" *25*, 162-163
  "2" *25*, 130, 195
Filter 184
  Pseudo-Filter 184
  für Window-System 195
findfifo, Funktion 249
Floating-Point-Exception 266
fopen, Standardfunktion 311
fork, Systemaufruf 117, 126–127, *138*, 139–141, 151–153, 161–162, 178, 181, 298
Formfeed 107
Fortran 18
fprintf, Standardfunktion 311
fputs, Standardfunktion 311
free, Standardfunktion 286, 291, 306, 312
freie Liste 26, 37, 303
fsck, Kommando 55
fsort, Funktion 190
fsort0, Programm 184, 186
fstat, Systemaufruf 67, 78, 90, 157, 202
Funktionen
  access 91
  assign 121
  builtin 179
  catchsig 274
  cget 101
  cleanup 274
  command 173
  cready 101
  dbmscall 222

Dclose 214
Dcreate 214
Ddelete 217
Dget 214
Dgetnext 216
Dopen 214
Dput 216
Dtop 215
edinvoke 190
edrcv 191
edsnd 191
entrysig 180, 269
EVexport 122
EVget 123
EVinit 123
EVprint 125
EVupdate 124
execlp2 152
execut2 140
execut3 182
execute 140
execvp2 151
find 120
fatal 17, 183, 277
findfifo 249
fsort 190
getln 95
getln2 96, 115
gettoken 169, 194
ignoresig 178, 269
invoke 174
itoa 160
jumper 277
keystroke 109
lock 28
lockpath 29
main 128, 136, 141
mainloop 275
mkdir2 114
mkfifo 209
msgctl 231
msgget 230
msgrcv 231
msgsnd 230
nullfcn 280
openfifo 206
openqueue 246
P 234–239, 252, 260, 286
rcvall 192
rcvsid 246
readany 100
receive 203–206, 213, 222, 229, 233, 235, 239, 246, 249, 252, 256, 260, 286, 296
redirect 180
refresh 103
restore 109, 284
rmqueue 206, 213, 222, 231
semtran 238, 241, 249
send 202, 213, 222, 229, 233, 235, 239, 246, 249, 252, 256, 260, 286, 296
setblock 98
setraw 111, 284
Sfdopen 97, 115
Sgetc 97
sndsid 246
Sopen 97
statusprt 146, 182
statutil 98, 276
syserr 17, 24, 165, 183, 272
timestart 300
timestop 300
turnaround 191
unlock 28, 30
V 234–239, 241, 252, 260, 286
waitfor 178, 182
Funktionstasten 93, 281
gemeinsam benützte Positionszeiger 13
gemeinsame Speicherbereiche 15, 242
gepufferte Ein- und Ausgabe 51
Gerätedatei 1, 4–5, 74, 91, 113, 303
/dev/null 181
Block-Gerätedatei 4, 303
Character-Gerätedatei 4
nackte 37, 57
Platten-Gerätedatei 294
Terminal-Gerätedatei 99, 195
zeichenorientierte 57
raw 37, 57
Gerätenummer 5, 74, 79, 295
Gerätetreiber 5, 57–58, 74, 261
für Terminals 286
kundenspezifische 93
Standard-Gerätetreiber 93
Terminal-Gerätetreiber 113
getc, Makro 312
getchar, Makro 194, 312
getegid, Systemaufruf 146
getenv, Standardfunktion 66, 118, 123, 312
geteuid, Systemaufruf 146
getgid, Systemaufruf 146
getgrgid, Standardfunktion 82, 312
getgrnam, Standardfunktion 82, 313
getln, Funktion 95
getln2, Funktion 96–97, 115
getlogin, Standardfunktion 50
getpgrp, Systemaufruf 146
getpid, Systemaufruf 27, 146, 300
getppid, Systemaufruf 146
getpwnam, Standardfunktion 82, 313

getpwuid, Standardfunktion 82, 313
gets, Standardfunktion 314
gettoken, Funktion *169*, 171, 173, 194
getuid, Systemaufruf 146
gewöhnliche Dateien 1–3, 5, 10, 53, 55, 57, 60, 67, 74, 90
  Ein- und Ausgabe 23
Gleitkommarechnung 263, 266
globale Shell-Variable 119
goto, C-Sprunganweisung 277
goto-Anweisung, in der Shell 194
Greenwich Mean Time 62
grep, Kommando 188
group leader 8
group-ID 9
Gruppe 9, 78
Gruppendatei, 9, 313
Gruppennummer 2, 8, *9*, 76, 79, 128, 304, 313
  effektive 9, 30, 69, 72, 76, 128, 147
  in einer Inode 10
  wirkliche 9, 72, 148
gtty, Systemaufruf 113
Hangup 8, 27, 138, *264*, 272
Hanser Verlag 1
Hardware-Interrupt 266
Hardware-Speicherverwaltung 292
Hardware-Register 18
highbyte, Makro 19, 145
Hintergrund-Kommando 179
  Eingabeumlenkung 181
Hintergrund-Prozeß 155
  cron 297
  abbrechen 278
HOME, Environment-Variable 182
I/O-Trap-Instruction 265
IBM
  IMS Datenbank 200
  MVS Betriebssystem 153, 200
ICANON-Bit 109, 111–112
ICRNL-Bit 106, 111–112
if-Anweisung, in der Shell 194
IGNCR-Bit 106
ignoresig, Funktion 178, 269
Ignorieren von Signalen 7, 128, *269*
IGNPAR-Bit 106
Iliste 60
Illegale Anweisung 265
impure 265
IMS, Datenbank von IBM 200
permutierter Index 75
init, Initialisierungsprozeß 7, 142–143, 278
INLCR-Bit 106, 111
Inode 2, 5, 13, 25–26, 53, 56–57, 60–61, 77–79, 303
  "1" 60
  Inhalt 2
instruction segment 6
int 15
Interrupt, Signal 8, 167, 174, 178–179, *265*, 269, 271–272, 275
Interrupt, Taste 7, 178, 186, 265, 272, 275
Inumber 2, 54–55, 60, 69, 79
  "2" 3, 150, 303
invoke, Funktion 174, 177–178, 180
ioctl, Kommando 113
ioctl, Systemaufruf 89, 93–94, *104*, 105, 107, 111, 113–114, 265, 271
ioctl.h, Definitionsdatei 113
IOT, PDP-11 Maschinenbefehl 265
IPC_CREAT-Bit 230, 238, 244
IPC_NDELAY-Bit 231
IPC_NOWAIT-Bit 230–231
IPC_RMID-Bit 231, 245
ISIG-Bit 108, 111
ISTRIP-Bit 106, 111
itoa, Funktion 160
IUCLC-Bit 106, 111
IXANY-Bit 107
IXOFF-Bit 107
IXON-Bit 107, 111
jumper, Funktion 277
Kalenderzeit 62
Kanal 58
kanonische Eingabe 109
Kapazität einer Pipe 156, 193
Katalog 1–3, 10, 26, 53–56, 67, 69, 71, 74, 304
  . . 150
  /dev 56, 74
  /etc 71
  /tmp 29, 71, 165
  /usr/bin 161
  /usr/include 300
  aktueller 3, 6
  Ausführungsrecht 26
  leerer 55
  lost+found 55
  Schreibzugriff auf 53
  Suchrecht 26
  untergeordneter 56
  wechseln 149
  Wurzelkatalog 3
Kataloghierarchie 71, 303
Katalogstruktur 69
Kennsatz, eines Dateisystems 294
Kern 6, 36, 39–40, 45, 57, 68, 74, 78
  Absturz 37
  Dateisystem 56
  Pufferung 39–41

keystroke, Funktion *109*, 112
kill, Kommando 278, 286
Kill, Signal 266
kill, Systemaufruf 264, 267, *277*, 278, 286–287
Kill, Zeichen 95, 98, 104
Kleinbuchstaben 106
Klient eines DBMS 198, 205
Kommando
  /bin/sh 132
  accton 304
  bc 195
  cat 164–165
  cd 149, 167, 293
  chmod 11
  chown 11
  cp 23
  date 302
  dc 195
  ed 271
  export 119, 125, 134–135, 152, 167
  find 69
  fsck 55
  grep 188
  ioctl 113
  kill 278, 286
  lex 168
  lint 270
  login 9, 147, 149–150
  ls 1, 10, 14, 54, 91
  mkdir 74–75
  mkfs 303
  mount 304
  mv 56, 69–71
  mvdir 56
  newgrp 9
  nice 138, 150
  nohup 8, 138, 152
  passwd 11
  pr 155
  ps 150
  pwd 74, 91
  rmdir 55
  set 125, 134–135, 167
  sort 155, 183–184, 186, 189
  stty 111–112, 114–115
    Option raw 111
    Option sane 112
  time 152, 300, 306
  touch 77, 90
  tty 113
  umask 293
  umount 304
  who 155
Kommandodatei 132
Kommandointerpreter 117, 119
Konsistenz einer Datenbank 256
Kontrollstrukturen der Shell 156, 194
Kontrollterminal *8*, 94, 264–265, 304
Koroutine 188
ld, Binder 291
leere Kataloge 55
Leitungsunterbrechung 264
Leserecht 26, 30, 91, 149
Lesezugriff
  auf Anweisungssegment 6
  auf Dateisystem 303
lex, Kommando 168
lexikalische Analyse 167
library 5
Linefeed 112
link, Systemaufruf 3, 56, *69*, 71, 73, 303
Linker 5
lint, Kommando 270
Liste, freie 26, 37, 60
LK_LOCK-Bit 259
LK_NBLCK-Bit 259
LK_NBRLCK-Bit 259
LK_RLCK 259
LK_RLCK-Bit 259
LK_UNLCK-Bit 259
loader 5
localtime, Standardfunktion 62, 66, 78, 82, 309, 314
lock, Systemaufruf 302
Lock-Manager
  bei DBMS 199, 259, 262
  in PC/IX 200
  in Xenix 3 200
lockf, Systemaufruf 258
locking, Systemaufruf *258*, 259–260
lockpath, Funktion 29
login, Kommando *9*, 147, 149–150
Login-Katalog 148
Login-Kommandodatei 293
Login-Name *9*, 50, 147, 313
  root 71
  Super-User 71
  Benutzer 50
logische Platte 56–57, 71, 294, 303
lokale Variable 119
longjmp, Standardfunktion *276*, 277, 286, 314, 316
lost+found, Katalog 55
lowbyte, Makro 19, 145
ls, Kommando 1, 10, 14, 54, 91
lseek, Systemaufruf 30–31, 34, *46*, 47–48, 50–51, 53, 95, 139, 157, 201, 259
main, Funktion 128, 136, 141–142
mainloop, Funktion 275

major-Nummer 74, 79
Makros
  getc 312
  getchar 194, 312
  highbyte 19, 145
  lowbyte 19, 145
  putc 315
malloc, Standardfunktion 121, 245, 270, 286, 289, 291, 306, 312, ***314***
Maschine, endliche 168
Maschinensprache 5
Maske für Dateianlage 26, 292
Maus 93
memcpy, Standardfunktion 249, 256, 315
Memory-mapped-Video 114–115
Menü-Shell 195
message.h, Definitionsdatei 209
Messages 12, 15, 155, 197–198, 200, 202, ***203***, 204–205, 209, 211, 218, 220, 222, 226, 229–230, 233, 235–236, 242, 252, 261, 263, 279
  älteste 231
  Implementierung mittels FIFO-Dateien 198, 202
  in BSD 4.2 262
  in System III 202
  in System V 198, 231, 236, 262, 296
  in Xenix 3 262
  Schlüssel 203
  Schlüsselnummer 230
  Typ 15, ***230***, 231, 236
  Werkzeuge 213
Message-Queue 15, 246, 263
  als Semaphor 235
minor-Nummer 74, 79
mkdir, Kommando 74–75
mkdir, Systemaufruf 74, 90
mkdir2, Funktion 114
mkfifo, Funktion 209
mkfs, Kommando 303
mknod, Systemaufruf 56, 67, 74–75, 202, 292
Montieren eines Dateisystems 294
mount, Kommando 304
mount, Systemaufruf 289, ***303***, 304–305
msgctl, Funktion 231
msgget, Funktion 230
msgget, Systemaufruf 246
msgrcv, Funktion 231
msgrcv, Systemaufruf 271, 279
msgsnd, Funktion 230–231
multiplexed-Dateien 90
Multitasking 101, 117
mv, Kommando 56, 69–71
mvdir, Kommando 56
MVS, IBM Betriebssystem 200
Nachrichten-Schlange 15
nackte Ein- und Ausgabe ***58***, 261
  für Terminals 111
nackte Gerätedateien 37, 57
nackte Platten-Gerätedateien 57, 60
nbwaitsem, Systemaufruf 241
Netzwerk 93
Neustart bei Stromausfall 267
newgrp, Kommando 9
Newline 98, 100, 106–107, 112, 174
nice, Kommando 138, 150
nice, Systemaufruf 12, ***150***
  in Version 7 152
nice-Wert 12, ***150***
no carrier, keine Verbindung 99
nohup, Kommando 8, 138, 152
Normalzeit 62, 66
nroff, Formatierer 13
nullfcn, Funktion 280
O_APPEND-Bit 32, 34, 48, 50–51, 99
O_CREAT-Bit 32–34
O_EXCL-Bit 32–34, 147, 235
O_NDELAY-Bit 32, 98–99, 156–157, 193, 201–202, 204, 231, 280
  in System III 286
  in Version 7 286
O_RDONLY-Bit 32
O_RDWR-Bit 32, 53
O_TRUNC-Bit 32–33, 53
O_WRONLY-Bit 32, 53
O_WSYNC-Bit 32, 37
Objektdatei 5
OLCUC-Bit 106
ONLCR-Bit 106
open, Systemaufruf 23–25, ***30***, 31–33, 35, 37, 48, 50, 53, 67, 74, 88–91, 94–95, 98–99, 113, 147, 157, 181, 201, 205, 230, 235, 241, 251, 271, 292
openfifo, Funktion 206
openqueue, Funktion 246
opensem, Systemaufruf 241
OPOST-Bit 111–112
Ortszeit 62, 314
owner group-ID 10
owner user-ID 10
P, Funktion 234–239, 241, 252, 260–261, 286
Paßwort 11, 71
Paßwortdatei ***9***, 28, 82, 147, 149, 313
page, Seite im Speicher 141
paging-Hardware 141
PARENB-Bit 106
parent-process-ID 7
Parität 106, 112
PARMRK-Bit 106
Pascal 17

passwd, Kommando 11
PATH, Environment-Variable 130, 132, 134
pause, Systemaufruf 33, 264, 271, ***279***, 280, 287, 316
PC/IX 296
  Bildschirmeditor EDIX 114
  curses-Paket 103
  FIFO-Dateien 198
  Lock-Manager 200
  lockf 258
  Record-Locking 198, 258
PDP-11 15, 197
  EMT Maschinenbefehle 265
  Gleitkomma-Hardware 265
  IOT Maschinenbefehle 265
  Segmentierungsregister 243
permutierter Index 75
Pfad 3, 78, 91
  absoluter 3, 60
  relativer 3
physikalische Platte 56
physikalische Schreibfehler 58
physikalische Ein- und Ausgabe 60
Pipe 25, 30, 78, 90, 201
  blockieren 156
  erzeugen 156
  Füllgrad 158
  Filedeskriptor 156–157, 159
  Kapazität 156, 193
  lesbares Ende 161, 163, 173–174, 177, 184, 188
  schreibbares Ende 160–161, 163, 173–174, 184, 186, 188
  Signale 266
pipe, Systemaufruf 20, 25, ***156***, 162, 165, 174, 177, 193, 202
Pipeline 155, 167, 266
  Ende-Symbol 174, 178
  ringförmige 155
  Zwei-Weg-Pipeline 183
Pipes 12–14, 35, 155, 202
  benannte 14
  bidirektionale 155–156
  Blockdepot 157
  Deadlock-Situation 158, 186, 188
  Ein- und Ausgabe 23
  gepufferte Ausgabe durch stdio 192
  in der Shell 155, 162
  Pufferung 157
  Verweiszahl 157
Platte 2, 56
  logische 56–57, 71, 303
  physikalische 56
Platten-Controller 46, 57
Platten-Gerätedatei 57, 60–61, 294
  nackte 57
Plattenfehler 37
plock, Systemaufruf 302
Positionierbewegungen 41, 48, 58
Positionszeiger 2, 12, 30, 34–35, 46–48, 50, 139
  stdio 190
  gemeinsam benützter 13
pr, Kommando 155
pread, Programm 159, 161–162, 164
printf, Standardfunktion 129, 311, 315, 317
Priorität 12, 128, 138, 150
process group 7
process's group-ID 8
process-group-ID 8
process-ID 7
profil, Systemaufruf 298
Programm ***5***, 6, 126
  überlagern 126
  Argument 117
  dbmstest 226
  envprint 118, 138
  fsort0 184, 186
  pread 159, 161–162, 164
  pwrite 159
  selbstmodifizierendes 6, 265
  statusprt 145
  Terminierung 268
Programmdatei 6, 68, 128
  eigene 17
  Ein- und Auslagern 68
Programmende 17
Programmfehler 265–266
Programmier-Konventionen 18
Programmiersprache 5
Programmname 304
Programmzähler 298
PROLOCK-Bit 302
Prompt, der Shell 136
Protokoll zum Zurücksetzen 256
Prozeß 1, ***6***, 8, 101, 126, 279, 304
  "0", Swap-Prozeß 278
  "1", init-Prozeß 278
  Ablaufverfolgung 265, 304
  Accounting 304
  Anweisungssegment 302
  Beschränkungen 290
  Datensegment 57, 302
  Ein- und Auslagern 7, 57, 139, 302
  Ende 39
  erzeugen 138
  Exit-Code 269, 274, 298
  im Speicher fixieren 302
  init 7
  initialisieren 127, 138

maximale Dateigröße 12
neu erzeugen 126
Priorität 150
sicher abbrechen 266
Signal-Reaktion 263
Status-Code 142–143
Swappen 7, 57, 139, 302
System-Datensegment 12
Terminierung 143
durch Signaleinwirkung 271
Trace-Flagge 12
Variablenexport 120
Waise 7
Zeitprofil erstellen 128, 298
Zombie-Prozeß 143, 274
Zugriff auf Environment 118
Prozeß-Tracing 12–12, *13*
Prozeßattribute
aktive Filedeskriptoren 128
Arbeitskatalog 128
Ausführungszeit 128, 139
Benutzernummer 304
Exit-Status 304
Gruppennummer 304
Kontrollterminal 304
Priorität 128
Prozeßgruppennummer 128
Prozeßnummer 128, 139
Vater-Prozeßnummer 128, 139
Vererbung 139
wirkliche Benutzernummer 128
wirkliche Gruppennummer 128
Wurzelkatalog 128
Prozeßende, anormal 266
Prozeßgruppe *7*, 264, 278
ändern 148
Prozeßgruppenleiter *8*, 264
Prozeßgruppennummer *8*, 9, 128
ändern 148
vererben 148
Prozeßkommunikation 21, 58, 155, 267, 298
in 4.2 BSD 260
in Version 7 260
mit Signalen 269
mittels kill 278
Prozeßnummer *7*, 9, 27, 128, 139, 148, 174, 277, 298
"0" 7
"1" 7
ausgeben 195
Eindeutigkeit 7, 147
Prozeßtabelle 143, 274
Prozeßzustand
bereit 302
blockiert 302
ps, Kommando 150
PS2, Environment-Variable 136
Pseudo-Filter 184
ptrace, Systemaufruf 265, *298*, 305
Puffer 4, 40, 97
des Dateisystems 36
entleeren 129, 297, 310
Pufferung
ausschalten 316
Benutzer 40
Kern 40
Pipes 157
pure-procedure 68
putc, Makro 315
pwd, Kommando 74, 91
pwrite, Programm 159
Quellprogramm 5
Queue 203
Eigentümer 231
Empfangsschlüssel 204
Kapazität 231
Zugriffsrecht 231
queue-ID 230
Quit, Signal 8, 167, 174, 178–179, *265*, 269, 271–272
Quit, Taste 178, 265
raw-Modus 111–114
rcvall, Funktion 192
rcvsid, Funktion 246
read, Systemaufruf 10, 15, 23–25, 30–31, *39*, 46–48, 57, 94–96, 98–101, 103, 109, 113, 156–158, 186, 193, 202–203, 231, 271, 279–281, 284–286
bei Dateiende 39, 47
Vorwärtsbewegung 50
readany, Funktion *100*, 103
Reaktion auf Signale 268
real group-ID 9
real user-ID 9
Real-Time 40, 46, 59–60, 299
realloc, Standardfunktion 122, 124, 286, 306, 312, 315
receive, Funktion 203–205, *206*, 213, 222, 229–231, 233, 235–236, 239, 246, 249–250, 252–253, 256, 260–262, 286, 296
receive, Systemaufruf 202
Record-Locking 5, 200, 242
Überschneidungen 259
bei /usr/group 198, 258
in PC/IX 198, 258
in Xenix 3 198, 256, 258–259
redirect, Funktion 180–181
refresh, Funktion 103
register, Vereinbarung 18

relativer Pfad 3
restore, Funktion ***109***, 111, 114, 284
Return 106–107, 112
rmdir, Kommando 55
rmqueue, Funktion ***206***, 213, 222, 231
root directory 3
root, Login-Name 11, 71, 149
round-robin-Methode 205
S_IFBLK-Bit 67
S_IFCHR-Bit 67
S_IFDIR-Bit 67
S_IFIFO-Bit 67
S_IFMT-Bit 67
S_IFREG-Bit 67
S_ISGID-Bit 67
S_ISUID-Bit 67, 70
S_ISVTX-Bit 67–68
Satzsperren, bei DBMS 242
save-swapped-text-Bit 67-68, 76
sbrk, Systemaufruf 244–245, 289–290, ***291***, 292, 306
Scheduler 12
Schlüssel bei Messages 203, 230
schnellere Ein- und Ausgabe 57
Schreiben 72
  verzögertes 36, 297
  physikalisches 58
Schreibrecht 26, 30, 77, 149
Schreibzugriff 40, 303
SD_CREAT-Bit 252
SD_NOWAIT-Bit 252
SD_RDONLY-Bit 251
SD_UNLOCK-Bit 252
SD_WRITE-Bit 251–252
sdb, Debugger 12, 264, 298
sdenter, Systemaufruf 252
sdget, Systemaufruf 251–252
sdgetv, Systemaufruf 253
sdleave, Systemaufruf 252
sdwaitv, Systemaufruf 252
seek, Systemaufruf 48
Segmentation-Violation 266
Segment 68
  bei Shared-Memory 242
Segmentgrenze 68
Segmentierungsregister 5, 114, 252
  bei PDP-11 243
Segmentierungsverletzung 6
Segmentseparierung 265
Seite, im Speicher 141
Semaphore 2, 12, 14–15, 27, 34, 186, 197–198, ***234***, 239, 242, 246, 262
  allgemeine 234
    in Xenix 3 242
  binäre 234–235,242
    in Xenix 3 240
  busy-wait 235
  Datei 27
  Dateiattribute 238
  Deadlock-Situation 242
  Eisenbahnwesen 14
  Erwerb 234, 238
  Freigabe 234
  Funktion P 234–239, 241, 252, 286
  Funktion V 234–239, 241–242, 252, 286
  in System V 198, 236
  in Xenix 3 15, 198, 236, 240–241
  Mechanismus 34
  Message-Queue 235
  Nummer 238
  Operationen 238
  Unteilbarkeit 237
  Unterstützung durch den Kern 235
  Variable 235
  Verwendung durch Super-User 27
  Zustände 234
semctl, Systemaufruf 238
semget, 237–238, 246
semop, Systemaufruf 237–238, 271
semtran, Funktion 238-239, 241, 249
send, Funktion 202–205, ***206***, 213, 222, 229–231, 233, 235–236, 239, 246, 249, 252–253, 256, 260–262, 286, 296
serialisierbar 256
set, Shell-Kommando 125, 134–135, 167
set-group-ID-Bit ***11***, 12, 26, 67, 76, 128
set-user-ID-Bit ***11***, 12, 26, 55, 67, 70, 75–76, 128, 303
set-user-ID-Programm 303
setblock, Funktion 98
setbuf, Standardfunktion 316
setgid, Systemaufruf ***147***, 148
setjmp, Standardfunktion ***276***, 277, 286, 314, 316
setpgrp, Systemaufruf 8, ***148***
setraw, Funktion ***111***, 112, 114, 284
setsig, Funktion 274
setuid, Systemaufruf ***147***, 148
Sfdopen, Funktion 97–98, 115
Sgetc, Funktion 97
sgtty.h, Definitionsdatei 113
Sharable-Instructions 68
Sharable-Text 68
Shared-Memory 12, 15, 197–198, 262, 286
  Zugriffsschutz 244, 252
  in BSD 4.2 262
  in System V 198, 242–243, 261–262
  in Xenix 3 15, 198, 242–243, 250–252, 261–262
  Segment 242

Shell 68
Abkömmlinge 9
abmelden 8
Anführungszeichen 167–168
Argumente in Anführungszeichen 155
Ausgabe umlenken 167, 173
Behandlung von Signalen 174
Kommandozeile prüfen 194
core dump 268
Ein- und Ausgabeumlenkung 155
einfaches Kommando 167, 173
Eingabe umlenken 167, 173
eingebautes Kommando 134, 167
cd 149, 167
export 134–135, 167
set 134–135, 167
Zuweisung 135, 167
Environment 119
Exit-Status 181
exportierte Variablen 119, 123, 125, 152
ausgeben 125
Fehlermeldungen 145, 181
Generierung von Dateinamen 156, 194
globale Variable 119
goto-Anweisung 194
Hintergrund-Kommando 179
Eingabeumlenkung 181
Hintergrundprozeß 155, 272
if-Anweisung 194
in Berkeley 4.3 BSD 194
in System V.2 194
cd 182
export 119, 125, 152
set 125
umask 293
Kommandodatei 132, 134
Kommandointerpreter 117, 119
Kommandozeile 165, 319
Kontrollstrukturen 156, 194
Leerzeichen 167–168
lokale Variable 119
Menü-Shell 195
Newline 167–168, 173–174
nichttödliche Fehler 183
Parameterersetzung 194
Pipe erzeugen 174
Pipeline 155, 167
Pipes 14, 155, 159, 162
Programm 119
Programme ausführen 127
Programmiersprache 119
Prompt-Zeichen 112, 136
Prozeßattribute setzen 139
Prozeßnummer ausgeben 182
Prozeßnummer von Hintergrundprozessen ausgeben 167
Rückwärtsschrägstrich 167
Reaktion auf Signale 179
Selbstmord 134
Semikolon 167–168, 173–174
set-user-ID-Bit 76
sh 139, 166, 183, 194, 266, 319
Signalbehandlung 269
Signale 178
Suche nach Programmdatei 130
Super-User-Shell 11, 76
Variablenwerte ausgeben 125
Vaterschaftsverhältnisse 194
wild cards 156, 194
Zeichenkette in Anführungszeichen 167
Zuweisungskommando 118–119, 121, 152
shmat, Systemaufruf *244*, 245, 270
shmctl, Systemaufruf 245
shmdt, Systemaufruf 245
shmget, Systemaufruf *244*, 245–246
SIG_DFL-Bit 269, 276, 284–285
SIG_IGN-Bit 269, 274
SIGALRM *267*, 279, 284–285
SIGBUS *266*, 268, 285
SIGCLD *267*, 269, 274, 285–286
SIGCLD-Bit 269
SIGEMT 266
SIGEMT 268, 285
SIGFPE *266*, 268, 278
SIGHUP 264, 268–269
SIGILL *265*, 268, 271
SIGINT 265, 268–269, 275
SIGIOT *265*, 268, 278, 285
SIGKILL *266*, 269
signal, Systemaufruf 12–13, 263–264, *268*, 269–270, 274, 276, 285, 287
Signale
Abbruch 266–267
Abfangen 7, 128, 143, *269*, 270, 276, 279, 285
Alarm-Clock 267
als Semaphor 263
Aufhören 265
Behandlung durch die Shell 174
Bus-Error 266
ed 275
Emulator-Trap-Instruction 266
erzeugen 157
exec 268–269
Für den Benutzer vereinbartes Signal "1" 267
Für den Benutzer vereinbartes Signal "2" 267
Fehler bei Gleitkommarechnung 266

Fehlerhaftes Argument in einem Systemaufruf 266
Floating-Point-Exception 266
Hangup 8, 27, 138, ***264***, 272
I/O-Trap-Instruction 265
ignorieren 7, 128, 263, 269
Illegale Anweisung 265
in Berkeley 4.2 BSD 264
Interrupt 8, 167, 174, 178–179, ***265***, 269, 271–272, 275
Kill 7, ***266***
Leitungsunterbrechung 264
Neustart bei Stromausfall 267
Prozeßabbruch 264
Prozeßkommunikation 269
Quit 8, 167, 174, 178–179, ***265***, 269, 271–272
Segmentation-Violation 266
SIGALRM ***267***, 279, 284–285
SIGBUS ***266***, 268, 285
SIGCLD ***267***, 269, 274, 285–286
SIGEMT 268, 285
SIGFPE ***266***, 268, 278
SIGHUP 264, 268–269
SIGILL ***265***, 268, 271
SIGINT 265, 268–269, 275
SIGIOT ***265***, 268, 278, 285
SIGKILL ***266***, 269
SIGPIPE 266
SIGPWR ***267***, 274, 285
SIGQUIT 268–269, 278
SIGSEGV ***266***, 268, 285
SIGSYS 266
SIGTERM ***266***, 267–268, 278
SIGTRAP ***265***, 268, 271, 298
SIGUSR1 267
SIGUSR2 267
Software-Termination 267
Standardreaktion 7, 128, 264, 267–268, ***269***, 271–272, 280
tödliches 143
Terminierung 263, 272
Tod eines Sohn-Prozesses 267
Trace-Trap 264, ***265***, 298
Unterbrechung vom Terminal 265
Unterbrechung von Systemaufrufen ***271***, 284, 286
Verletzung der Segmentierung 266
vom UNIX-Kern 263
Warteschlange 286
Weck-Signal 271, 280, 284
Wecker 267, 279, 281, 284–285
Zurücksetzen 271
SIGPIPE 266
SIGPWR ***267***, 274, 285
SIGQUIT 265
SIGQUIT 268–269, 278
SIGSEGV ***266***, 268, 285
SIGSYS 266
SIGTERM ***266***, 267, 268, 278
SIGTRAP ***265***, 268, 271, 298
SIGUSR1 267
SIGUSR2 267
sigvec, Systemaufruf 286
sleep, Standardfunktion 29, 33, 101, 139, 279–280, 287, 316
sleep, Systemaufruf 280
sleep2 287
sndsid, Funktion 246
socket-Dateien 90
Software-Termination 267
Sohn-Prozeß 6, 11–12, 25, 139, 141–142, 149, 152, 184, 186, 274, 279, 293, 298, 302, 319
Terminierung 143, 267
Sommerzeit 62–63, 66
Sopen, Funktion 97
sort, Kommando 155, 183–184, 186, 189
Special-File 1, 4, 10, 14, 35
Ein- und Ausgabe 23
Speicher 68
Speicherauszug, core dump 143, 265, 268, 278
Speicherklasse
auto 18
Speicherverwaltung
dynamische 314
Hardware-unterstützte 292
virtuelle 141, 291
Sperrdatei 50
Sperre 256
exklusive 256
gemeinsame 256
sprintf, Standardfunktion 160, 316
sscanf, Standardfunktion 317
Stack 275, 286, 314
Standard, /usr/group 114
Standard-Ausgabe ***25***, 94, 129, 173, 195, 315
Standard-Bücherei der Ein- und Ausgabe, stdio 41, 45, 97, 129, 142, 158, 190, 192
Standard-Eingabe ***25***, 94, 97, 135, 164, 168, 173, 195, 312, 314
Standardfunktionen
abort 265
asctime 62, 82, 309, 314
atoi 160, 309
atol 310
fflush 310
fclose 142, 310

fdopen 98, 190, 310
fgets 310
fopen 311
fprintf 311
fputs 311
free 291, 306, 312
getenv 66, 118, 123, 312
getgrgid 312
getgrnam 313
getlogin 50
getpwnam 313
getpwuid 313
gets 314
localtime 62, 66, 82, 309, 314
longjmp ***276***, 277, 286, 314, 316
malloc 121, 245, 270, 286, 289, 291, 306, 312, 314
memcpy 249, 256, 315
printf 129, 311, 315, 317
realloc 122, 124, 286, 306, 312, 315
sbrk 291
setbuf 316
setjmp ***276***, 277, 286, 314, 316
sleep 29, 33, 101, 139, 279–280, 287, 316
sprintf 160, 316
sscanf 317
strcat 317
strchr 134, 317
strcmp 317
strcpy 318
strcspn 124, 318
strlen 318
strncpy 318
strrchr 73, 319
strtok 135, 319
system 319
tmpnam 27
ungetc 169, 320
Standardpaket
ctime 62–63, 66
stat, Systemaufruf 61, 67, 70, ***78***, 79, 90–91, 202, 295
statische Variablen 291
Status-Code ***142***, 143
Statusänderungszeitstempel 77, 79
Statusinformation 78
statusprt, Funktion 146, 182
statusprt, Programm 145
statutil, Funktion 79, 98, 276
stdio 41, 45, 97, 129, 142, 158, 190, 192
sticky-Bit 68
stime, Systemaufruf 301
strcat, Standardfunktion 317
strchr, Standardfunktion 134, 317
strcmp, Standardfunktion 317
strcpy, Standardfunktion 318
strcspn, Standardfunktion 124, 318
stream.h, Definitionsdatei 41
strlen, Standardfunktion 318
strncpy, Standardfunktion 318
strrchr, Standardfunktion 73, 319
strtok, Standardfunktion 135, 319
struct termio 104
stty, Kommando 111–112, 114–115
Option sane 112
stty, Systemaufruf 94, 113
Suchrecht 10, 72, 149
Super-Block 60, 91, 294, 303
Super-User 3–4, 11–12, 27, 33, 55–56, 69–71, 73–74, 76–77, 90, 278, 294
Accounting 305
alle Prozesse abbrechen 278
Dateisystem demontieren 304
Dateisystem montieren 303
einbrechen 71
Grenze für Dateigröße erhöhen 290
Identität wechseln 147
nice-Wert erhöhen 150
Prozeß im Speicher fixieren 302
Systemzeit setzen 301
Wurzelkatalog ändern 149
Zugriff auf Semaphor-Datei 33
svc 15
SVID, System V Interface Definition 291
Swap-Device 68
Swap-Prozeß 278
Swap-Space 68
Swappen eines Prozesses 7, 57, 139
sync, Systemaufruf 289, ***297***
sys 15
sys3b, Systemaufruf 305
sys_errlist, externe Variable 16
sys_nerr, externe Variable 16
syserr, Funktion 17, 24, 165, 183, 272
system data segment 6
System III 21, 198, 299
FIFO-Dateien 197
Messages 202
multiplexed-Dateien 90
O_NDELAY-Bit 280, 286
Portabilität 89, 296
Prozeßkommunikation 12
Systemaufrufe 20
Signal SIGCLD 274
Signal SIGPWR 274
Signaltypen 264
Standard-Gerätetreiber 94
Systemaufrufe
creat 30
dup2 89, 193

fcntl 193
open 30
utime 77
write 37
System V
accton, Kommando 304
COFF, Common Object File Format 5
Dateiattribute 241
Definitionsdatei /usr/include/times.h 299
Interface Definition, SVID 291
Messages 198, 236, 262
multiplexed-Dateien 90
Portabilität 89, 296
Prozeßattribute 307
Prozeßkommunikation 12, 20, 59
Semaphore 198, 236
Deadlock-Situation 242
Shared-Memory 198, 242–243, 261–262
Signal SIGCLD 274
Signal SIGPWR 274
Signaltypen 264
Standard-Gerätetreiber 94
Standardfunktionen
memcpy 249, 315
Systemaufrufe
creat 30
dup2 89, 193
fcntl 193
msgget 246
plock 302
semctl 238
semget 237–238, 246
semop 237–238
shmat ***244***, 245, 270
shmctl 245
shmdt 245
shmget ***244***, 245–246
utime 77
system, Standardfunktion 319
System-CPU Zeit 299
System-Daten 6
System-Datensegment 6, 12, 126–128, 138–139, 143, 279
System-Time 40, 46, 59–60
Systemadministrator 47, 62, 71, 73, 76, 91
Systemaufrufe
access 71–73, 91
alarm 33, 264, 267, ***279***, 280, 285–286, 316
brk 244–245, 290, ***291***, 292
chdir 149
chmod 26, 67, 76, 79, 88, 292
chown 76, 79
chroot 3, ***149***, 150, 152
chsize 2
close 23–24, 30, ***39***, 94, 98, 113, 129, 157, 163
creat 23–24, ***25***, 27, 29–33, 35, 48, 53, 56, 67, 74, 89, 95, 157, 235, 241, 292–293
creatsem 241
dup 25, 88–89, 157, ***162***, 163–165, 181, 193–194
dup2 89, 193
exec 88, 117, ***126***, 127, 130, 134, 138–139, 141, 148, 153, 161–163, 178, 181, 269, 279, 298, 302
execl ***127***, 128–130, 131, 132, 161
execle 131
execlp ***131***, 132, 152, 161
execv ***131***, 132
execve 131
execvp 131, 132, 137–138, 140–141, 151–152, 161, 173
exit 16, ***142***, 143, 181, 267, 269, 274
fcntl 25, ***88***, 89–90, 94, 98–99, 104, 130, 156–157, 163, 193–194, 200
fork 117, 126–127, ***138***, 139–141, 151–153, 161–162, 178, 181, 298
fstat 67, 78, 90, 157, 202
getegid 146
geteuid 146
getgid 146
getgrgid 82
getgrnam 82
getpgrp 146
getpid 27, ***146***, 300
getppid 146
getpwnam 82
getpwuid 82
getuid 146
gtty 113
ioctl 89, 93–94, ***104***, 105, 107, 111, 113–114, 265, 271
kill 264, 267, ***277***, 278, 286–287
link 56, 69, 71, 73, 303
lock 302
lockf 258
locking ***258***, 259–260
lseek 30–31, 34, ***46***, 47–48, 50–51, 53, 95, 139, 157, 201, 259
mkdir 74, 90
mknod 56, 67, 74–75, 202, 292
mount 289, ***303***, 304–305
msgget 246
msgrcv 271, 279
nbwaitsem 241
nice 12, ***150***, 152
open 23–25, ***30***, 31–33, 35, 37, 48, 50, 53, 67, 74, 88–91, 94–95, 98–99,

open 113, 147, 157, 181, 201, 205, 230, 235, 241, 251, 271, 292
opensem 241
pause 33, 264, 271, ***279***, 280, 287, 316
pipe 20, 25, ***156***, 162, 165, 174, 177, 193, 202
plock 302
profil 298
ptrace 265, ***298***, 305
read 10, 15, 23–25, 30–31, ***39***, 46, 48, 50, 57, 94–96, 98–101, 103, 109, 113, 156–158, 186, 193, 202–203, 231, 271, 279–281, 284–286
receive 202
sbrk 244–245, 289–290, ***291***, 292, 306
sdenter 252
sdget 251–252
sdgetv 253
sdleave ***252***, 253
sdwaitv ***252***, 253
seek 48
semctl 238
semget 237, ***238***, 246
semop 237–238, 271
setgid ***147***, 148
setpgrp 8, ***148***
setuid ***147***, 148
shmat ***244***, 245, 270
shmctl 245
shmdt 245
shmget ***244***, 245–246
signal 264, ***268***, 270, 274, 276, 285, 287
sigsem 241–242
sigvec 286
sleep 280
stat 61, 67, 70, ***78***, 79, 90–91, 202, 295
stime 301
stty 94, 113
sync 289, ***297***
sys3b 305
time 62, ***301***
times 62, ***299***, 300, 305
ulimit ***289***, 292
umount 304
uname ***295***, 296, 305
unlink ***26***, 27, 31, 55–56, 69, 71, 73, 256
ustat 294
utime 77, 79, 88, 90
vfork 141, 153
wait 117, 141–142, ***143***, 146, 186, 269, 271, 274, 279, 286, 298–299
waitsem 241
write 10, 20, 23–25, 30–31, 34 ***35***, 36–39, 46–48, 50–51, 53, 57, 94, 98, 113, 156–158, 186, 193, 202–203, 271
Systemkonfiguration 289–290
Systemprozeß 278
Systemzeit
  bestimmen 301
  setzen 301
Tab 106, 107
TAB3-Flagge 107
Taschenrechner, dc 195
TCFLSH-Bit 105
TCGETA-Bit 105
TCSBRK-Bit 105
TCSETA-Bit 105
TCSETAF-Bit 105
TCSETAW-Bit 105
TCXONC-Bit 105, 107
temporäre Datei 28, 147, 165
Terminal
  abhängen 263, 264
  Ausgabe 97–98
  Bildschirmterminal 114
  blockierungsfreie Ein- und Ausgabe 98
  Blockierung 98
  Eingabe 286
  Kontrollterminal 94, 264–265
  normale Ein- und Ausgabe 94
  programmierbares 93
  raw-Modus 111, 114
  zum Schreiben öffnen 95
Terminal-Gerätedatei 97, 99, 195
  wirklicher Name 94
  /dev/tty 94
Terminal-Gerätetreiber 113, 286
Terminal-Parameter
  Baud-Rate 106
  Echo 108
  Parität 106
  Stop-Bits 106
  Zeichen-Wiedergabe 108
  Zeichensatz 106
Terminalinformation 104
  Ausgabeflußsteuerung 107
  Eingabeflußsteuerung 107
  Flußsteuerung 107
  Parität 112
  Steuerzeichen 107
  Tabulator 107
  verzögerungsfreie Eingabe 109, 112
  Zeichengröße 106
  Zeichensatz 106
  Ein- und Ausgabe 93
Terminalsteuerung 20
Terminaltreiber 93, 104, 115
Terminierung 272
Terminierungssignal 7
text segment 6, 68

impure 265
text-busy 68
Textdatei 5
tick, Zeiteinheit 299
time, Kommando 152, 300, 306
time, Systemaufruf 62, ***301***
times, Systemaufruf 62, ***299***, 300, 305
Timesharing 285
timestart, Funktion 300
timestop, Funktion 300
tmpnam, Standardfunktion 27
touch, Kommando 77, 90
Trace-Flagge 12
Trace-Trap 264, ***265***, 298
Trace-Trap-Modus, Ablaufverfolgung 143
Transaktion 256
tty, Kommando 113
tty-group ID 8
turnaround, Funktion 191
TXTLOCK-Bit 302
TZ, Environment-Variable 62–63, 66, 118
Übersetzer 5
Übertragungsende, end-of-transmission 95
ulimit, Systemaufruf ***289***, 292
umask, Kommando 293
umount, Kommando 304
umount, Systemaufruf 304
uname, Systemaufruf ***295***, 296, 305
undo-Protokoll 256
ungetc, Standardfunktion 169, 320
UNIX
Dateisystem 40, 303
Derivat 132
Filter 25
Implementierung 35, 37
Kern 62, 101, 142, 244–245, 297, 302
Absturz 303
Accounting-Information schreiben 304
Blockdepot 4, 297
copy-on-write-Verfahren 141
Dateisystem 93, 303
Lock-Manager 200
Scheduler 12
Signale 263
Starten 127
Unterstützung von Semaphoren 235
Name 295
Portabilität 19
Sicherheitsproblem 73
Version 37, 40, 295
unlink, Systemaufruf ***26***, 27, 31, 55–56, 69, 71, 73, 256
unlock, Funktion 28, 30
UNLOCK-Bit 302
Unteilbarkeit 14, 35
Unterbrechung von Systemaufrufen 271, 284, 286
untergeordnete Kataloge 56
user data segment 6
user-ID 9
User-Time 40, 46, 59–60
ustat, Systemaufruf 294
utime, Systemaufruf 77, 79, 88, 90
V, Funktion 234–239, 241–242, 252, 260–261, 286
Variable
effektive Benutzernummer 147
end 291
environ 118–120, 123, 128, 130
errno 16, 29, 33, 37, 45, 98, 143, 165, 218, 241, 272, 279
sys_errlist 16
sys_nerr 16
Vater-Katalog 72
Vater-Prozeß 6, 11–13, 25, 120, 139, 141–142, 149, 184, 186, 266, 269, 271, 279, 298
Vater-Prozeßnummer 7, 9, 128, 139, 142
Vaterschaftsverhältnisse, in der Shell 194
Verletzung der Segmentierung 266
Version 7 21
close-on-exec-Flagge 89
DBMS 261
fortgeschrittene Prozeßkommunikation 260
multiplexed-Dateien 90
O_NDELAY-Bit 286
Portabilität 50, 89
Prozeßgruppe 286
raw-Modus 111
Signaltypen 264, 285
Systemaufrufe
chroot 152
creat 30
dup2 193
execlp 152
execvp 152
fcntl 89, 163
gtty 113
ioctl 113
kill 286
lock 302
nice 152
stty 113
Versionsnummer, eines UNIX-Systems 295
Verweis 3, 10, 13, 25–26, 55–56, 69, 71, 79, 303
Verweiszahl 4, 26, 78–79
bei Pipes 157
verzögertes Schreiben 36, 297
verzögerungsfreie Eingabe 109, 112

vfork, Systemaufruf 141, 153
vi, Bildschirmeditor 103, 280–281, 286
virtuelle Speicherverwaltung 141, 291
VMS, DEC Betriebssystem 153
void, Datentyp 17, *18*, 268
vollkommener Baum 69, 71
vorsichtige Austauschtechniken 37, 51
Waise 7
Waisendateien 55
wait, Systemaufruf 117, 141–142, *143*, 146, 186, 269, 271, 274, 279, 286, 298–299
waitfor, Funktion 178, *182*
waitsem, Systemaufruf 241
Warteschlange 203, 286
Weck-Signal 6, *267*, 271, 279–285
who, Kommando 155
wild cards, in der Shell 156, 194
Window-System 195
wirkliche Benutzernummer *9*, 71–72, 128, 148, 278
wirkliche Gruppennummer *9*, 72, 128, 148
write, Systemaufruf 10, 20, 23–25, 30–31, *35*, 36–39, 46–48, 50–51, 53, 57, 94, 98, 113, 156–158, 186, 193, 202–203, 271
  dazwischenliegende Blöcke 47
  hinter Dateiende 47
Wurzel-Dateisystem 91
Wurzelkatalog 3, 60, 128, 150, 303, 304
  ändern 149
XCASE-Bit 106
Xenix 3
  curses-Paket 103
  FIFO-Dateien 198
  Lock-Manager 200
  memcpy, Standardfunktion 256
  Messages 262
  O_WSYNC-Bit 32, 37, 50
  Prozeßkommunikation 59
  Record-Locking 198, 256, 258–259
  Semaphore 15, 198, 236, 240–241
    allgemeines 242
    binäres 240
    Deadlock-Situation 242
  Shared-Memory 15, 198, 242–243, *250*, 251–252, 261–262
  Systemaufrufe
    chsize 2
    creatsem 241
    lock 302
    locking *258*, 259–260
    nbwaitsem 241
    sdenter *252*
    sdget 251, *252*
    sdgetv 253
    sdleave *252*, 253
    sdwaitv *252*, 253
    sigsem 241–242
    waitsem 241
Zeichen
  Backspace 106–107
  Control-J 112
  Control-Q 107
  Control-S 107
  Control-X 98
  EOL 109
  EOT 95, 97–98, 104, 109, 112, 115
  Erase 98, 104
  Formfeed 107
  Kill 98 104
  Linefeed 112
  Newline 95, 97–98, 100, 106–107, 112
  Rückwärtsschrägstrich 106
  Return 95, 106–107, 112
  Steuerzeichen 107
  Tab 106
  Tabulator 107
Zeichenkette 73
Zeichenumwandlung 106
Zeilenende 96
Zeilenpufferung, Ausgabe auf Terminal 129
Zeiten
  Ausführungszeiten 62, 128, 139
  Benutzer-CPU 299
  Kalenderzeit 62
  Normalzeit 62
  Ortszeit 62, 314
  Real-Time 40, 46, 59, 299, 304
  Sommerzeit 62
  Startzeit 304
  System-CPU 299
  System-Time 40, 46, 59, 304
  Systemzeit 301
  User-Time 40, 46, 59, 304
  verbrauchte Gesamtzeit 304
  Vergleiche 63
  Zone 62
Zeitangabe 309, 314
  tick 299
zeitempfindliche Algorithmen 285
Zeitmessung 300
Zeitprofil erstellen 128, 298
Zeitstempel 77–78
  Statusänderung 2, 62
  Änderung 2, 62, 77
  Zugriff 2, 62, 77
Zeitzone 62
zentralisiertes DBMS 199
Zombie-Prozeß 143, 274

Zugriffsmuster 39
Zugriffsrecht
    Benutzer 9
    Besitzer 72
    Gruppe 72
    Queue 231
    Welt 72
Zugriffsschutz *10*, 24, 59, 67, 78, 199, 292
    Ausführungsrecht 10, 149
    Benutzernummern
    effektive Benutzernummer 127–128, 147, 278
    effektive Gruppennummer 128, 147, 278
    Leserecht 10, 149
    Shared-Memory 244, 252
    Schreibrecht 10, 149
    set-group-ID-Bit 128
    set-user-ID-Bit 128, 303
    Suchrecht 149
    wirkliche Benutzernummer 148, 278
    wirkliche Gruppennummer 148, 278
Zugriffszeitstempel 11, 79
Zuweisung, Shell-Kommando 121, 135, 167
Zwei-Weg-Pipeline 183

Hanser

FUNDIERTE
FACHBÜCHER
KOMPETENTER
AUTOREN

Neue Fachbücher für UNIX und Programmiersprache C

Ammon/Fröhlich
C-Tools
Werkzeuge für die Programmierung in C. Von R. von Ammon und M. Fröhlich. Herausgegeben vom ECO-Institut, Lappersdorf. 360 Seiten. 1987. Gebunden.
ISBN 3-446-14974-0

Bolsky
C tabellarisch
Ein AT&T Programmierhandbuch. Von M. I. Bolsky. Die deutsche Ausgabe besorgten Dr. E. Janich und Dipl.-Math. T. Mandry. 128 Seiten. 1987. Spiralheftung.
ISBN 3-446-14977-5

Feuer
Der C-Trainer
Von A. R. Feuer. Übersetzt von R. von Ammon. Coedition Carl Hanser Verlag/Prentice-Hall International. Etwa 250 Seiten. 1987. Kartoniert.
ISBN 3-446-15055-2

Kernighan/Pike
Der UNIX-Werkzeugkasten
Programmieren mit UNIX. Von B. W. Kernighan und R. Pike. Die deutsche Ausgabe des legendären Werkes „The UNIX Programming Environment" besorgte Prof. Dr. A. T. Schreiner. 418 Seiten, 21 Tabellen. 1987. Kartoniert.
ISBN 3-446-14273-8

Kernighan/Ritchie
Programmieren in C
Mit dem C-Reference Manual in deutscher Sprache. Von B. W. Kernighan und D. M. Ritchie. Reihe: PC professionell – Arbeitsplatzrechner in Ausbildung und Praxis. Die deutsche Ausgabe besorgten Prof. Dr. A. T. Schreiner und Dr. E. Janich. 275 Seiten, C-Syntax-Graphen. 1983. Kartoniert.
ISBN 3-446-13878-1

Schreiner
Professor Schreiners UNIX-Sprechstunde
Technik, Tips und Tricks von awk bis yacc. Von Prof. Dr. A. T. Schreiner. Etwa 160 Seiten mit 48 Programmbeispielen und Lösungen. 1987. Kartoniert.
ISBN 3-446-14894-9

Volck
Software-Engineering mit UNIX-Workstations
Von Dipl.-Ing. R. Volck. Reihe: PC professionell – Arbeitsplatzrechner in Ausbildung und Praxis. Etwa 240 Seiten. 1987. Kartoniert.
ISBN 3-446-14985-6

Carl Hanser Verlag
Postfach 860420
8000 München 86
Telefon (089) 92694-0

HANSER

Hanser
FUNDIERTE
FACHBÜCHER
KOMPETENTER
AUTOREN

UNIX-Handbuch zur Programmentwicklung

Bach/Domann/
Weng-Beckmann (Hrsg.)
UNIX
Handbuch zur Programmentwicklung.

Herausgegeben von Dipl.-oec. Fred Bach, Dipl.-Ing., Dipl.-Wirtsch.-Ing. Peter Domann und Dipl.-Inf. Ulrike Weng-Beckmann, alle Siemens AG, München. Autoren: Dipl.-Math. A. Baur, Dipl.-Ing., Dipl.-Wirtsch.-Ing. P. Domann, H. Folters, E. Geiger, Dipl.-Math. U. Loyda, K. Luks, Dipl.-Inf. V. Meyer, Dipl.-Inf. G. Spies, Dipl.-Inf. U. Weng-Beckmann, P. Wiedemann. 452 Seiten. 1987. Gebunden. ISBN 3-446-15103-6

Die mit UNIX angebotenen Werkzeuge erfüllen - richtig eingesetzt - die Anforderungen, wie sie heute in industriellen Softwareprojekten gestellt werden. Was jedoch in der reichhaltigen UNIX-

Literatur fehlt, sind geeignete Anleitungen, die den Entwickler beim gezielten Einsatz der UNIX-Werkzeuge im Rahmen des modernen Software-Engineering unterstützen. Diese Lücke möchte das vorliegende Buch schließen.

Es beschreibt den Einsatz von UNIX-Werkzeugen im Rahmen des Software-Engineering und aus der Sicht der strengen Anforderung, die man heute an den industriellen Entwicklungsprozeß stellt.

Es entstand aus der langjährigen Erfahrung, welche die Autoren beim Einsatz von UNIX in unterschiedlichen Projekten gewonnen haben. Dabei haben sie erkannt, wo Probleme beim Einsatz von UNIX-Werkzeugen entstehen und wie man typische Fallen umgehen und Fehlerquellen vermeiden kann. Diese Erfahrung soll mit dem Handbuch weitergegeben werden. Es will dazu beitragen, daß die unter UNIX entwickelte Software dem Qualitätsstandard kommerzieller Produkte entspricht.

Carl Hanser Verlag
Postfach 860420
8000 München 86
Telefon (089) 92694-0

HANSER

Das Thema dieses Buches sind UNIX-Systemaufrufe, die die Schnittstellen zwischen den Benutzerprogrammen und dem UNIX-Kern bilden, auf dem sie ablaufen. Für UNIX Programmierer sind grundlegende Kenntnisse über die Systemaufrufe unentbehrlich, denn sie sind die einzige Möglichkeit, die Multitasting-Mechanismen oder die elementaren Einrichtungen zur Prozeßkommunikation zu benutzen. Ziel dieses Buches ist es daher, jedem erfahrenen Programmierer die Möglichkeit zu bieten, UNIX-Systemaufrufe leicht zu lernen und sie dann vernünftig und portabel einzusetzen. Deshalb enthält dieses Buch auch eine Fülle von Beispielen: über 3500 Zeichen C-Code.

Marc J. Rochkind, der Entwickler des berühmten Systems zur Versionskontrolle, SCCS, zeigt in diesem Buch beispielhaft die Verwendung grundlegender Systemaufrufe zur Dateibehandlung, zum Terminalzugriff, wie auch alle Möglichkeiten der Prozeßkommunikation bis hin zur zentralisierten Datenbank. Außerdem durchleuchtet er noch Signale und verschiedene nützliche Systemaufrufe.

Die vollständige Behandlung der Systemaufrufe macht dieses Buch einzigartig. Es ist deshalb für jeden Systemprogrammierer ein unentbehrliches Handwerkszeug und eine notwendige Ergänzung des UNIX-Handbuches.

ISBN 3-446-15202-4